Blazor és társai

kliensoldali webfejlesztés JavaScript helyett C#-ban, WebAssembly felett

Mózs Barnabás

2020.05.24.

Tartalomjegyzék

Feladat leírása

A féléves feladatom fő célja modern kliensoldali webfejlesztéshez használt technológiák megismerése, kipróbálása és összehasonlítása volt. Ezen megismert technológiák közös eleme, hogy magasszintű programozási nyelveket támogatnak és ezen nyelveken írt kódot WebAssembly nyelvre fordítják, ezzel alternatívát nyújtva a JavaScript alapú webfejlesztés mellett.

A két technológia, amivel a félév során foglalkoztam, a Microsoft által fejlesztett Blazor és az nventive, illetve nagyrészt közösség által fejlesztett Uno Platform volt. A megismerésükhöz és összehasonlításukhoz egy olyan webalkalmazás megvalósítása volt a cél, ami a Wikidata által tárolt adatokat képes általános, valamint az adatok egyes tulajdonságaitól függő formában megjeleníteni.

Fontos szempont volt, hogy a két különböző technológiával készült alkalmazás közös C# nyelven írt logikát használjon az adatok kezeléséhez és kinézetükben, illetve funkcióikban is a lehető legjobban megegyezzenek. Az elkészült feladat megfelel ezen szempontoknak, azonban az alkalmazások csak az általános megjelenítési módra képesek. Habár a feladat ilyen szempontból befejezetlen, mégis alkalmas a technológiák ismertetésére és többszempontbeli összehasonlítására.

Főbb technológiák

WebAssembly

A WebAssembly egy bináris szintaktikát használó alacsony szintű programozási nyelv, amely nem tartalmaz hardverspecifikus elemeket. A fejlesztésének célja a böngészőben történő, biztonságos és hatékony kliensoldali szkriptelés, valamint nem titkoltan a weben egyeduralkodó JavaScript alapú fejlesztés leváltása.

A WebAssembly bájtkódot egy verem alapú virtuális gép futtatja a böngészőben, amely egyben a hardverplatformon elérhető optimalizációkért is felel. A szabványt mára az össze nagyobb böngésző támogatja és számos a használatára épülő komolyabb keretrendszer fejlesztése is megkezdődött.

Blazor

A Blazor egy Microsoft által fejlesztett, nyílt forráskódú, webfejlesztést támogató keretrendszer, ami lehetőséget biztosít olyan alkalmazások fejlesztésére, melyek logikája a

megszokott JavaScript helyett C# nyelven lett implementálva. Ennek segítségével lehetővé teszi a .NET-es alkalmazáslogikák közvetlen használatát webes környezetben is. A felületek leírására a html egy kiterjesztett változata használható.

Két különböző alkalmazástípus fejlesztését támogatja, a kliensoldali, illetve szerveroldali webalkalmazását. A szerveroldali megoldás az ASP.NET Core-os alkalmazásszerveren dinamikusan renderelt tartalmakra épül, míg a kliensoldali a WebAssembly szabványt használja.

Uno Platform

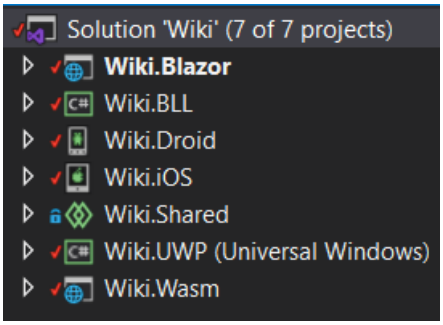
Az Uno Platform egy nyílt forráskódú multiplatform keretrendszer, aminek segítségével egyidejűleg fejleszthető közös C# nyelven írt logikát használó, hasonló kinézetű UWP, Andoroid, iOS és WebAssembly szabványt használó webalkalmazás. Az alkalmazások felületének leírására a XAML nyelv használható.

Wikidata

A Wikidata a Wikimédia Alapítvány egyik projektje, amely egy szabad, kollaboratív, többnyelvű adatbázis. Strukturált adatok gyűjtésével támogatja a Wikipédiát, és a többi Wikimédia projektet, illetve bárkit az egész világon.

Megvalósítás

A feladat megvalósítását a megfelelő projektek létrehozásával kezdtem egy közös Wiki nevű solution-be. Az eredmény az 1. ábrán látható hét projektet tartalmazó solution lett.



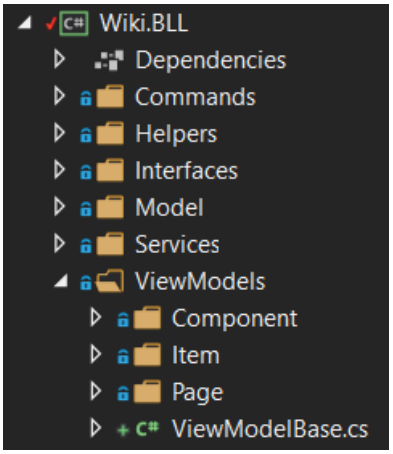
1. ábra - Wiki solution

A Wiki.Blazor projektet a Blazor Webassembly App template-el hoztam létre. Ezt egy Components és egy Pages nevű mappával egészítettem az alkalmazás vezérlőinek rendszerezett tárolásához.

Az Uno Platfrom-hoz tartozó projekteket egy Uno Platform Solution Template nevű extension segítségével generáltam. A Cross-Platform App template öt darab porjektet hozott létre. A Wiki.UWP, Wiki.Droid, Wiki.iOS és Wiki.Wasm projektek az Uno által támogatott különböző platformokhoz tartozó alkalmazások, a Wiki.Shared pedig az általuk használt közös kódot tartalmazó projekt. Ebbe a Blazor-hoz hasonlóan létrehoztam a Pages mappát, illetve

később további mappákat, amik a megfelelő UI elemeket és vezérlőtípusokat tárolják, mint például UserControls vagy DataTemplates.

A közös alkalmazás logika megvalósításához egy Wiki.BLL nevű .NET Standard projektet hoztam létre a 2. ábrán látható struktúrával. Kezdetben csak a Services, Interfaces, Helpers és ViewModels mappákat tartalmazta.



2. ábra - Wiki.BLL projekt

A Services mappába kerültek a WikiData adatainak kezeléséért felelős szolgáltatás osztályok, az Interfaces-be pedig az általuk megvalósított interfészek.

A Helpers mappa kezdetben csak a közös Dependency Injection-t lehetővé tevő statikus `ServiceConfigHelper` osztályt tartalmazta, amely kiterjesztő függvényeket valósított meg az `IServiceCollection` interfészhez. A mappa tartalma a későbbiekben tovább bővült a fejlesztést segítő kiterjesztő függvényeket megvalósító osztályokkal.

A ViewModels mappa az MVVM minta megvalósításához szükséges ViewModel osztályok rendezett tárolására szolgál. A feladat megoldásához nem használtam már létező, az MVVM minta használatát támogató keretrendszert, mivel szerettem volna saját ViewModel rendszert fejleszteni, hogy részben ezzel is megismerkedjek a félév során.

A ViewModeleket Component, Item és Page alkategóriákba soroltam. Az ItemViewModelek feladata a konkrét megjelenítéshez szükséges adatok kezelése, míg a Component- és PageViewModeleké a különálló komponensek és oldalak által megjelenítendő adatokat kezelő ItemViewModelek tárolása és koordinálása, valamint a távoli adatok kezeléséért felelős szolgáltatás osztályok használata.

Mivel a ViewModeleknek az állapotváltozásuk értesítéséhez implementálniuk kell `INotifyPropertyChanged` interfészt, ezért létrehoztam számukra egy közös absztrakt `ViewModelBase` ősosztályt, ami az alábbi módon segíti a konkrét ViewModelekben a megfelelő property-k létrehozását.

```
protected T Get<T>(ref T field) => field;

protected void Set<T>(ref T field, T value, [CallerMemberName] string propertyName = "") where T :
class
{
    if (field != value)
    {
        field = value;
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
```

```
    }
}
```

A ViewModelekben ezután már elég volt az alábbi property implementáció, amihez Code Snippet-et is létrehoztam, hogy gyorsítsam a fejlesztést.

```
public string Label
#region get; set;
{
    get => Get(ref _label);
    set => Set(ref _label, value);
}
private string _label = string.Empty;
#endregion

public string Description
get; set;
```

Kezdetben úgy terveztem, hogy az alkalmazás két oldalból fog állni. Egy `SearchPage`-ből, amin kereshetők a Wikidata által tárolt entitások, illetve egy `EntityPage`-ből ami a megjeleníti a `SearchPage`-en kiválasztott entitás adatait. Ehhez létrehoztam egy `SearchPageViewModel`-t, ami a `SearchServices` szolgáltatásosztályt használta, hogy egy beírt szöveges érték alapján, megjelenítse azon entitások címkéjét egy listában, amiknek a címkéje tartalmazza a megadott értéket.

Ehhez a szolgáltatásban először a Wikidata SPARQL végpontját használtam. Létrehoztam egy paraméterezhető SPARQL lekérdezést egy string-ként, ami azokat az entitásokat adja vissza, amiknek a label értéke tartalmazza a megadott paramétert. A kereséskor ezt a lekérdezést küldtem el a végpontnak egy http get hívásban. Azonban ez nagyon lassúnak bizonyult, így végül áttértem a MediaWiki API használatára.

MediaWiki API használatakor probléma merült fel Cross-Origin Resource Sharing miatt. Eleinte ezt a problémát egy cors-anywhere nevű proxy segítségével oldottam meg, majd később egy saját proxy-t implementáltam egy ASP.NET Core-os WebAPI formájában, aminek az egyetlen végpontja egy get kérést fogad és küld az alábbi módon.

```
[HttpGet("{url}")]
public async Task<string> GetAsync(string url)
       => await _httpClient.GetStringAsync(Uri.UnescapeDataString(url));
```

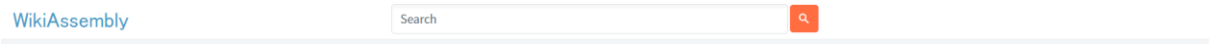
Hogy az alkalmazás minden kérést ezen keresztül küldjön, a ServiceConfigHelper-ben az alábbi módon regisztráltam be a HttpClient-et.

```
private static Uri _corsApiUrl = new Uri("https://localhost:5001/api/cors/");
. . .
public static IServiceCollection Configure(this IServiceCollection services)
{
    services.AddSingleton(new HttpClient { BaseAddress = _corsApiUrl });
    . . .
    return services;
}
```

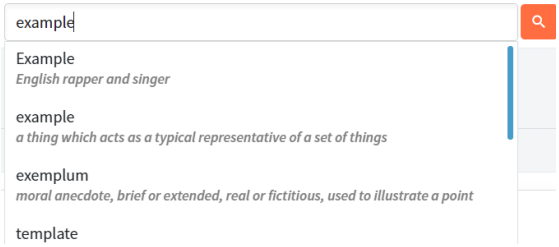
A MediaWiki API wbsearchentities végpontjának használata jóval hatékonyabbnak bizonyult, mint a SPARQL-es megoldás.

A `SearchPage`-et később feleslegesnek találtam, így töröltem és a keresés funkciót egy `SearchComponentViewModel`-be helyeztem át, amit az alkalmazás fejlécében megvalósított keresősávban használtam fel.

A Blazor alkalmazás felületét kezdetben próbáltam alap html elemekből felépíteni, majd a megjelenésüket css-ben stílusozni. A teszteléshez ezek megfeleltek, azonban külsejükben nem nyújtottak kellően jó megoldást. Hogy szép és egységes felületeket hozzak létre, a Radzen Blazor Components nevű könyvtárat használtam fel. A könyvtárban található RadzenMenu és AutoComplete vezérlőket felhasználva az alkalmazás fejlécét az alábbi módon valósítottam meg.



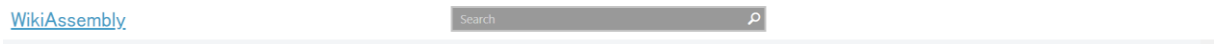
Az AutoComplete-nek az `oninput` eseményére elküldtem a wbsearchentities végpontnak a beírt szöveget és a visszakapott adatokat `SearchResultItemViewModel-ek` formájában átadtam a vezérlőnek, ami azt egy listában jelenített meg az alábbi módon.



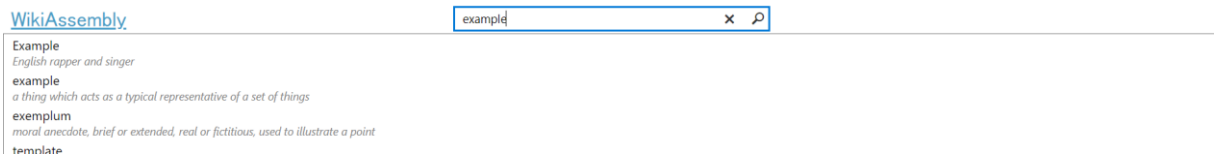
A Radzen sajnos nem támogatta a listaelemek testre szabását, így eleinte csak a találatok címkéjét tudtam megjeleníteni. Később elértem a leírásuk megjelenítését is JavaScript segítségével az alábbi módon.

```
<RadzenAutoComplete Data="@viewModel.SearchResultItemViewModels"
                    . . .
                    LoadData="@SetSearchDescription" />
. . .
private async Task SetSearchDescription()
{
    await Task.Delay(1);
    await JSRuntime.InvokeVoidAsync("setSearchDescription",
                                    "ui-autocomplete-items",
                                    viewModel.SearchResultItemViewModels.Select(c => c.Description)
                                                                        .ToArray());
}
. . .
function setSearchDescription(text, list) {
    var elemetsByClass = document.getElementsByClassName(text);
    . . . // textnode-ok létrehozása a list tartalma alapján, majd azok listaelemekhez való hozzáfűzése
}
```

Az Uno esetében a fejlécet egy NavMenu nevű UserControl-ba implementáltam, ami valójában egy Grid konténert használ az elemek elhelyezéséhez.



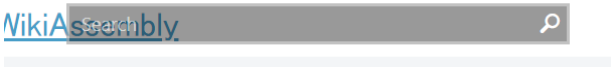
A keresés megvalósításához az `AutoSuggestBox` vezérlőt használtam.



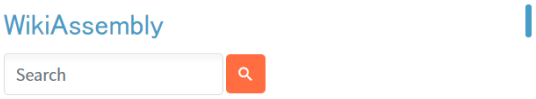
Az Uno esetén a feladat ezen része sokkal könnyebb volt, mivel az `AutoSuggestBox` támogatja a listaelemek testre szabását, viszont a Blazor-os megoldásnak köszönhetően legalább kipróbáltam a JavaScript ilyen módú használatát is.

Az Uno-os megoldásban nem szabtam testre a fejléc megjelenését stílusozással, így az a változó ablakméretet nem követi megfelelően. A Blazor-ös megoldás esetén erre külön figyeltem.

Uno



Blazor



Ennek az implementálásakor felmerült az a probléma, hogy a Radzen-es vezérlők stílusának külön fájlba szervezése nehézkes és gyakorlatilag csak az alábbi módon lehetséges.

```
<RadzenMenu Attributes="@(new Dictionary<string, object> { { "class", "nav" } })">
    . . .
</RadzenMenu>

.nav {
    display: flex;
    . . .
}
```

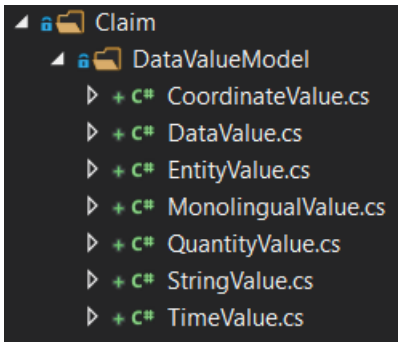
Ez a megoldás azontúl, hogy kényelmetlen, előfordult, hogy elrontotta a vezérlő működését.

A kiválasztott entitások lekéréséhez, azok azonosítóját és a MediaWiki API wbgetentities végpontját használtam. Az adatokat json formátumban kaptam meg. A feldolgozásukhoz tanulmányoznom és reprodukálnom kellett a hozzájuk tartozó adatmodellt. Ez nem okozott különösebb nehézséget és az alábbi módon oldottam meg.

```
public class Entity
{
    public string Title { get; set; }

    public string Id { get; set; }
```

```
    public Dictionary<string, Label> Labels { get; set; } = new Dictionary<string, Label>();

    public Dictionary<string, Description> Descriptions { get; set; } = new Dictionary<string,
Description>();

    public Dictionary<string, List<Alias>> Aliases { get; set; } = new Dictionary<string,
List<Alias>>();

    public Dictionary<string, List<Claim>> Claims { get; set; } = new Dictionary<string,
List<Claim>>();

    public Dictionary<string, SiteLink> SiteLinks { get; set; } = new Dictionary<string, SiteLink>();

}
```

A `Claim` struktúra megalkotásakor felmerült, hogy a `Claim`-hez tartozó `Snak` több típusú is lehet, és a típusinformációt a `Snak` `DataValue`-jának `Type` property-je tárolja. Hogy támogassam a különböző típusokat a 3. képen látható struktúrát hoztam létre és az alábbi módon értem el, hogy a json parse-oláskor a megfelelő típusú `DataValue` jöjjön létre.



3. ábra - DataValue struktúra

```
public class Snak
{
    private DataValue _dataValue;
    public DataValue DataValue
    {
        get => _dataValue;

        set
        {
            if (_dataValue == null)
            {
                _dataValue = value.CastToSpecific();
            }
        }
    }
}

public class DataValue
{
    [JsonProperty("value")]
    public object RawValue { get; set; }

    public string Type { get; set; }

    public DataValue CastToSpecific()
    {
        if (Type == EntityValue.Type)
        {
            return JsonConvert.DeserializeObject<EntityValue>(RawValue.ToString());
        }
        else if (Type == StringValue.Type)
        {
            return new StringValue { Value = RawValue.ToString() };
        }
        . . .
        return this;
    }

    public virtual string GetValue() => RawValue.ToString();
```

```
public class EntityValue : DataValue
{
    public static new readonly string Type = "wikibase-entityid";

    public string Id { get; set; }

    public override string GetValue() => Id;
}
```

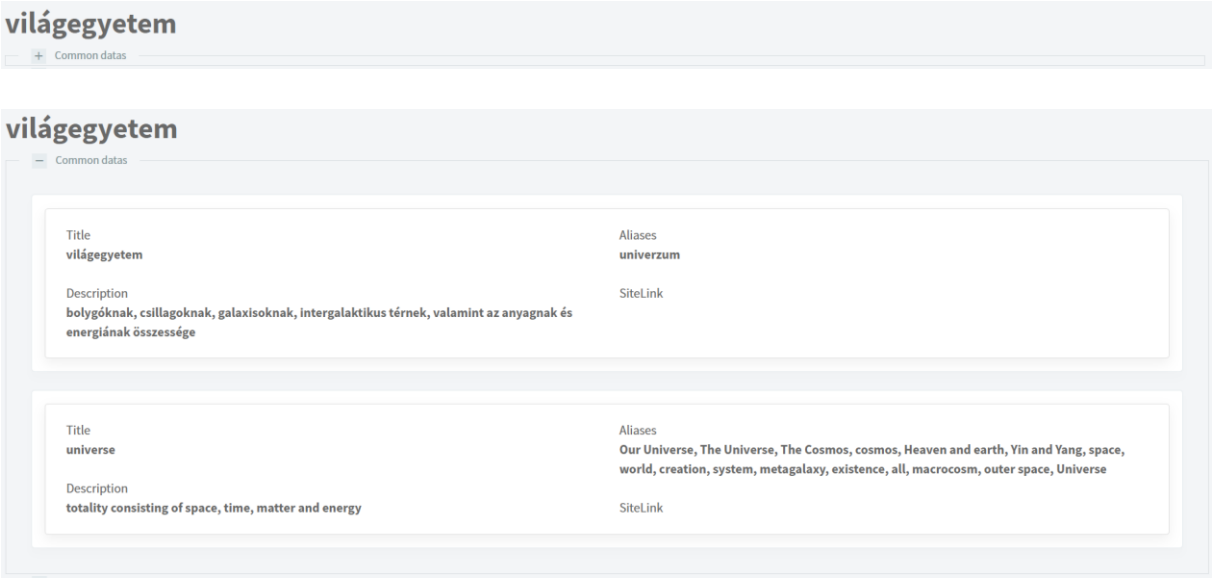
```
}
```

A `Claim`-ek kezelésekor felmerült az a probléma is, hogy az állítmányok és az entitás típusú `DataValue`-k valójában további Wikidata entitások, amik címkéjének kiíratásához a keresett entitáshoz hasonlóan a MediaWiki API wbgetentities végpontja segítségével le kell kérdezni az adataikat. Szerencsére a végpont támogatta egyszerre több, de legfeljebb ötven darab entitás lekérdezését. A problémát az alábbi módon oldottam meg.

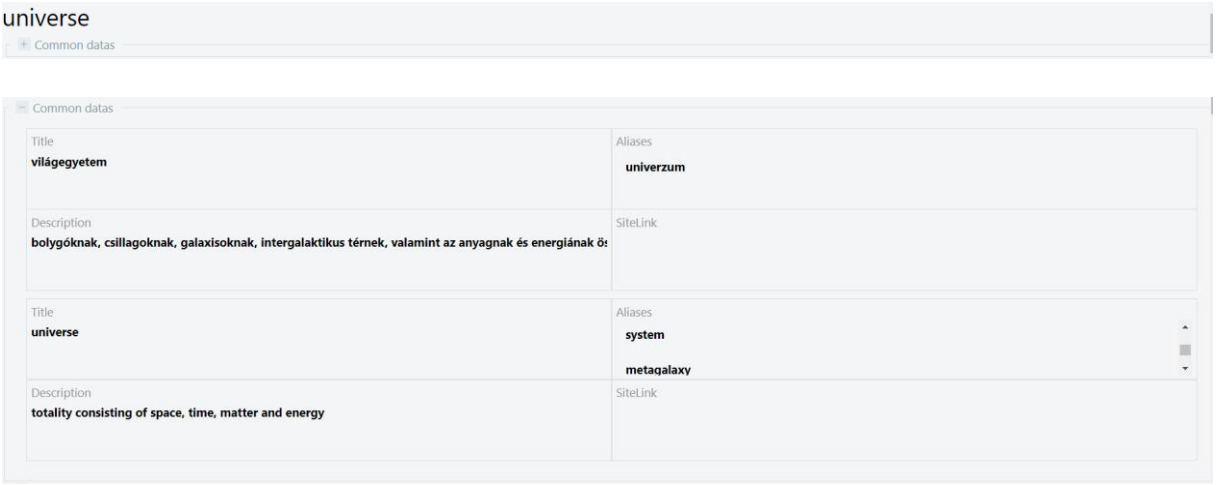
```
public async Task<Dictionary<IClaimData, List<IClaimData>>>
    LoadClaimsAsync(Entity entity, List<string> languages = null)
{
    List<string> idParameters = CreateIdParameters(GetIdsFromClaims(entity.Claims));

    var entities = await GetClaimEntites(idParameters,
        async (idParameter) =>
            await _httpClient.GetStringAsync(GetUrl(idParameter,
                                                    languages,
                                                    "labels".ToStringList())));
    return GetClaims(entity.Claims, entities);
}

private List<string> GetIdsFromClaims(Dictionary<string, List<Claim>> claims)
{
    var ids = new List<string>();
    . . .
    return ids.Distinct()
              .ToList();
}

private List<string> CreateIdParameters(List<string> idList)
{
    var idParameters = new List<string>();
    for (int i = 0; i < idList.Count; i += _parameterLenghtLimit)
    {
        idParameters.Add(GetListParameter(idList, i, idList.Count - i));
    }

    return idParameters;
}

private string GetListParameter<T>(List<T> list,
                                   int startIndex = 0,
                                   int limit = 0) =>
    list?.GetRange(startIndex,
                   Math.Min(_parameterLenghtLimit,
                            limit < 1 ?
                            list.Count :
                            limit))
    ?.ToString("|");

public static string ToString<T>(this List<T> list,
                                 string separator = ", ")
=> string.Join(separator, list);
```

```
public class EntityService : ServiceBase, IEntityService
{
    private const int _parameterLenghtLimit = 50;
    . . .
}
```

A Blazor alkalmazásban a keresett entitás általános adatainak megjelenítéséhez `RadzenFieldset` és `RadzenDataList` vezélőket, valamint alap html elemeket használtam. A megoldás az alábbi módon néz ki.

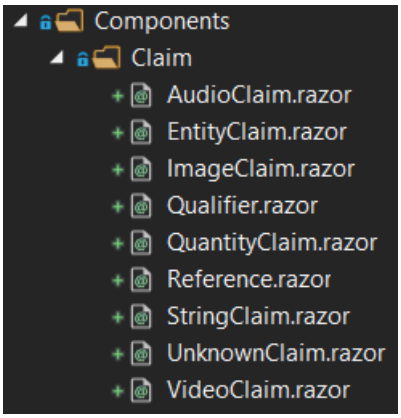


Az Uno esetén próbáltam hasonló kinézetet és működést elérni egyedi UserControl vezérlők segítségével az alábbi módon.



A `Claim`-ek megjelenítése már bonyolultabbnak bizonyult, hiszen a különböző típusú `Snak`-ekhez különböző megjelenítés szükséges. A Blazor esetén a megjelenítést `RadzenFieldset` és `RadzenGrid` segítségével, valamint saját komponensek létrehozásával oldottam meg az alábbi módon.

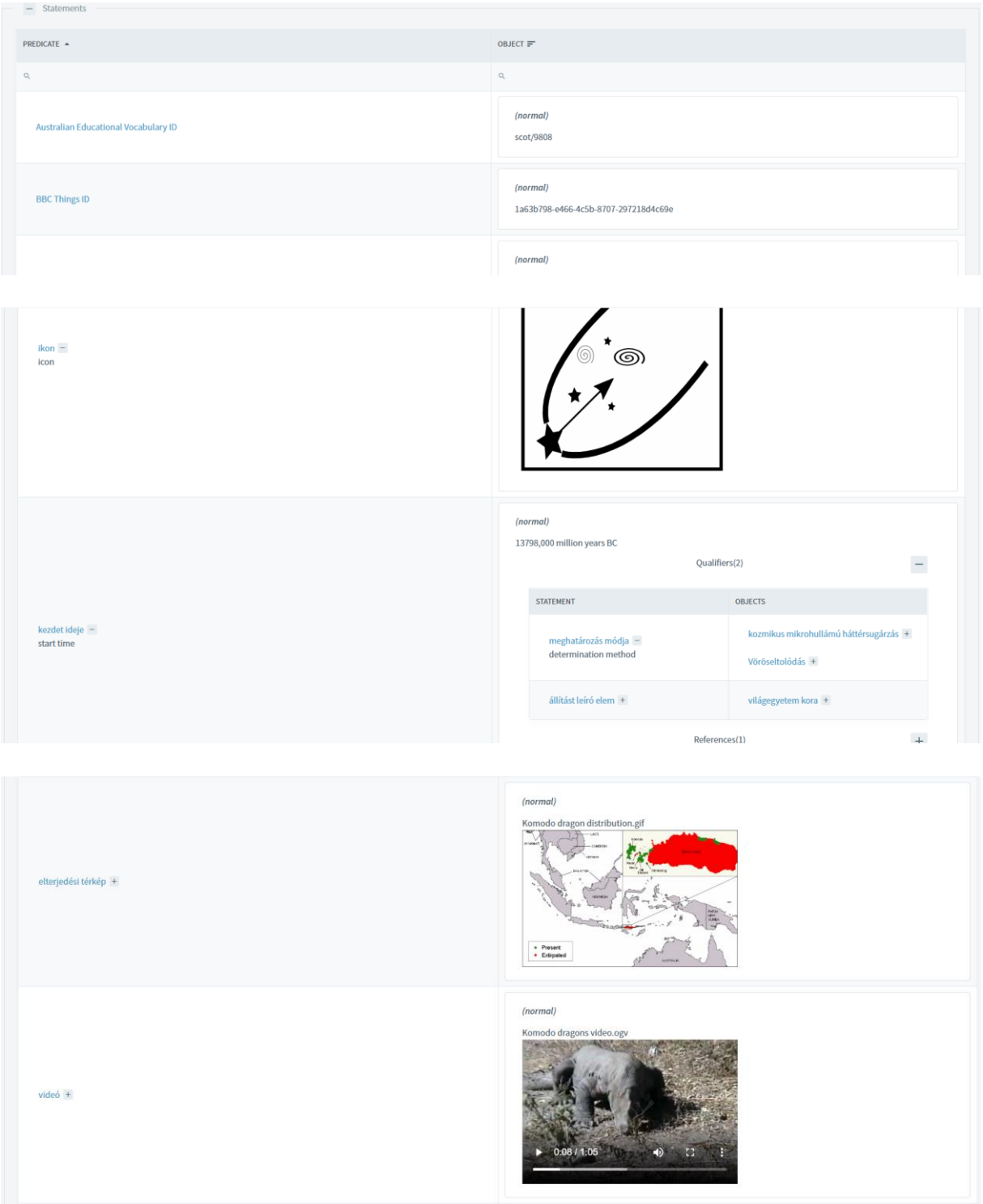
```
<RadzenFieldset . . .>
    . . .
    <ChildContent>
        <RadzenGrid . . .>

            <Columns>
```




```
                <RadzenGridColumn . . .>

                    <Template Context="claimItemViewModel">

                @GetSpecificComponent(claimItemViewModel.Predicate)

                    </Template>

                </RadzenGridColumn>

                <RadzenGridColumn . . .>

                    <Template Context="claimItemViewModel">
                        @foreach (var claimData in claimItemViewModel.Objects)
                        {
                            <Radzen.Blazor.RadzenCard>

                                <h6 class="rank">(@claimData.Rank)</h6>

                                @GetSpecificComponent(claimData)

                                @if (claimData.Qualifiers.Any())
                                {
                                    <Wiki.Blazor.Components.Claim.Qualifier
Qualifiers="claimData.Qualifiers"

GetSpecificComponent="GetSpecificComponent" />
                                }

                                @if (claimData.References.Any())
                                {
                                    <Wiki.Blazor.Components.Claim.Reference
References="claimData.References"

GetSpecificComponent="GetSpecificComponent" />
                                }

                            </Radzen.Blazor.RadzenCard>
                        }
                    </Template>
                </RadzenGridColumn>
            </Columns>
        </RadzenGrid>
    </ChildContent>
   </RadzenFieldset>

private RenderFragment GetSpecificComponent(ClaimDataItemViewModel claimData) => builder =>
{
    ClaimDataItemViewModel viewModel = new UnknownClaimDataItemViewModel();
    ComponentBase component = new Components.Claim.UnknownClaim();
    if (claimData is EntityClaimDataItemViewModel entityClaimData)
    {
        component = new Components.Claim.EntityClaim();
        viewModel = entityClaimData;
    }
    else if (claimData is StringClaimDataItemViewModel stringClaimData)
    {
        component = new Components.Claim.StringClaim();
        viewModel = stringClaimData;
    }
    . . .
    builder.OpenComponent(0, component.GetType());
    builder.AddAttribute(1, "ViewModel", viewModel);
    builder.CloseComponent();
```

```
};
```

A megoldás az alábbi módon néz ki.



A megoldás egy rendezhető és szűrhető, több nyelvű adatokat, valamint audió és videó tartalmakat megjeleníteni képes táblázat lett.

Az Uno esetén ezt a feladatot egy kisebb, a `Claim`-ek megjelenítésére szolgáló UserControl struktúra létrehozásával, `DataTemplateSelector`-al és a `Snak` típusoknak megfelelő `DataTemplatek`-ek implementálásával oldottam meg az alábbi módon.

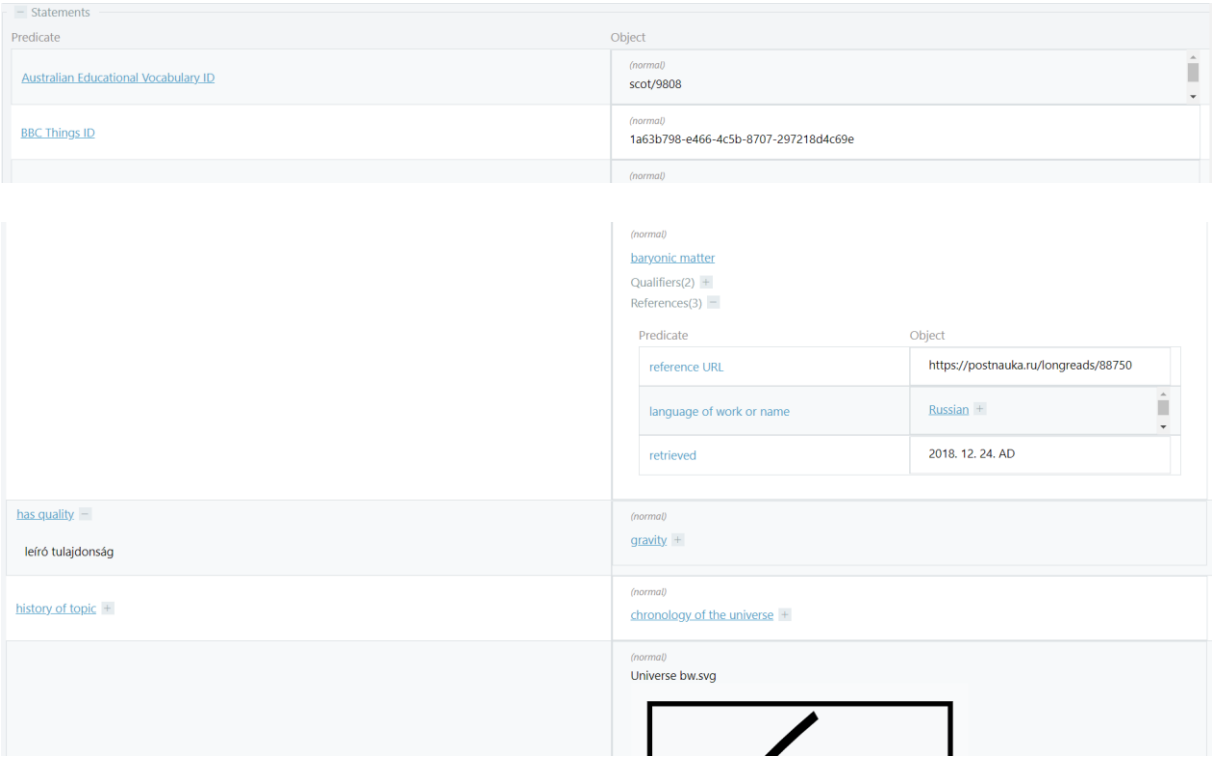
```
<usercontrols:Fieldset.InnerContent>

    <StackPanel Orientation="Vertical">

        <Grid . . .>
            . . .

            <TextBlock Grid.Column="0"
                       Text="Predicate"
                       Foreground="{StaticResource Gray}"
                       PointerPressed="SortByPredicate"/>
            . . .
        </Grid>

        <ListView ItemsSource="{x:Bind viewModel.EntityItemViewModel.Claims}"
                  . . .>
            . . .
            <ListView.ItemTemplate>
                <DataTemplate x:DataType="viewmodels:ClaimItemViewModel">

                    <Grid>
                        . . .
                        <usercontrols:Panel . . .">

                            <usercontrols:Panel.Header>

                                <HyperlinkButton . . ./>

                            </usercontrols:Panel.Header>

                            <usercontrols:Panel.InnerContent>

                                <TextBlock . . ./>

                            </usercontrols:Panel.InnerContent>

                        </usercontrols:Panel>
                        . . .
                        <ListView . . .>

                            <ListView.ItemTemplateSelector>
                                <claimdata:ClaimDataTemplateSelector
                                    Entity="{StaticResource EntityClaimDataTemplate}"
                                    String="{StaticResource StringClaimDataTemplate}"
                                    Quantity="{StaticResource QuantityClaimDataTemplate}"
                                    Image="{StaticResource ImageClaimDataTemplate}"
                                    Video="{StaticResource VideoClaimDataTemplate}"
                                    Audio="{StaticResource AudioClaimDataTemplate}"
                                    Unknown="{StaticResource UnknownClaimDataTemplate}"/>
                            </ListView.ItemTemplateSelector>

                            <ListView.ItemContainerStyle>
                                . . . <!-- Kék és fehér cellák váltakozása -->
                            </ListView.ItemContainerStyle>

                        </ListView>
                    </Grid>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </StackPanel>
</usercontrols:Fieldset.InnerContent>
```

```
public sealed partial class EntityPage : Page
{
    private readonly EntityPageViewModel
viewModel;
    . . .
    private void SortByPredicate(object sender,
PointerRoutedEventArgs args) =>
        viewModel.EntityItemViewModel.Claims
= new ObservableCollection<ClaimItemViewModel>(
viewModel.EntityItemViewModel.Claims.OrderBy(c
=> c.PredicateSortProperty));
    . . .
}
```

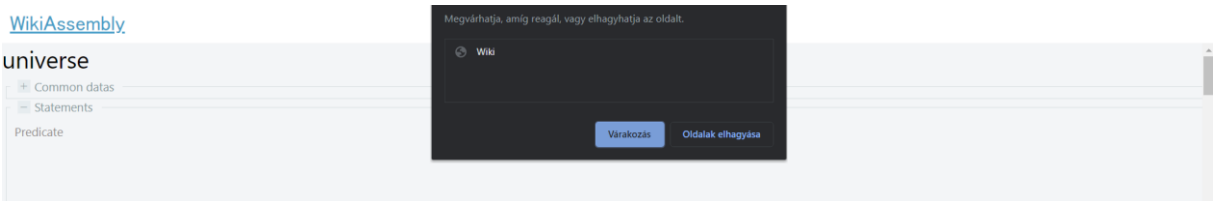
```
public class ClaimDataTemplateSelector : DataTemplateSelector
{
    public DataTemplate Entity { get; set; }
    public DataTemplate String { get; set; }
```

```
    . . .
    protected override DataTemplate SelectTemplateCore(object itemViewModel)
    {
        if(itemViewModel is EntityClaimDataItemViewModel)
        {
            return Entity;
        }
        else if (itemViewModel is StringClaimDataItemViewModel)
        {
            return String;
        }
        . . .
        else
        {
            return Unknown;
        }
    }
}

<!-- Entity -->
<DataTemplate x:Key="EntityClaimDataTemplate"
              . . .>
    <usercontrols:ClaimObject . . .>
        <usercontrols:ClaimObject.InnerContent>
            <usercontrols:Panel . . .>
                <usercontrols:Panel.Header>

                    <HyperlinkButton . . ./>

                </usercontrols:Panel.Header>
                <usercontrols:Panel.InnerContent>

                    <TextBlock . . ./>

                </usercontrols:Panel.InnerContent>
            </usercontrols:Panel>
        </usercontrols:ClaimObject.InnerContent>
    </usercontrols:ClaimObject>
</DataTemplate>

<!-- String -->
<DataTemplate x:Key="StringClaimDataTemplate"
              . . .>
    <usercontrols:ClaimObject . . .>
        <usercontrols:ClaimObject.InnerContent>

            <TextBlock . . ./>

        </usercontrols:ClaimObject.InnerContent>
    </usercontrols:ClaimObject>

</DataTemplate>
```

A megoldás az alábbi módon néz ki.



Sajnos ez megoldás csak a szöveges és képi tartalmak megjelenítését támogatja. A megvalósítása sokkal összetettebb volt és több utánajárást igényelt, mint a Blazor esetén. Illetve előkerült az a probléma, hogy habár az alkalmazás a háttérben már betöltötte a megjelenítendő adatokat, a felület összeállítása nagyon sok időt vesz igénybe és nem ritka a böngésző alábbi üzenete.



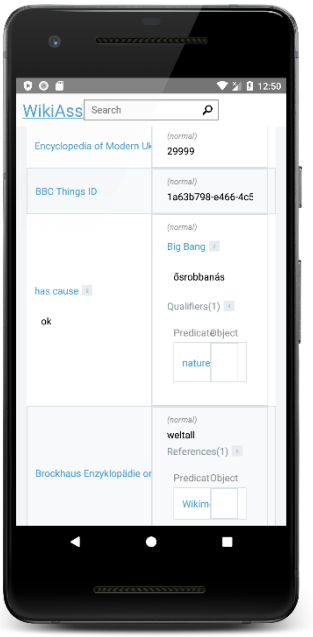
Összegzés

A feladatot és a technológiák megismerését nagyon izgalmasnak találtam. A Blazor összeségében jobb benyomást keltett webalkalamazás fejlesztése kapcsán, mint az Uno Platform, habár fontos megjegyezni, hogy az Uno elsődleges lényege a multiplatform fejlesztés, illetve nem áll mögött egy olyan multinacionális cég, mint a Microsoft.

Véleményem szerint a Blazor-os client-side alkalmazás fejlesztés jelenlegi állapotában már teljesen alkalmas lehet, akár komolyabb megjelenésű alkalmazások fejlesztésére is. A felületek

ilyen módú html leírása és css stílusozása jól működik és komponens könyvtárak használatával nagymértékben meggyorsítható a fejlesztés.

Az Uno Platform alkalmas lehet hasonló kinézetű multiplatform alkalmazások fejlesztésére, habár a webalkalamzás esetén, a lassú felületösszeállítás egy ponton túl nem teszi lehetővé az összetett felületek használatát. A fejlesztés során kifejezetten a webes alkalmazás fejlesztésével foglalkoztam és nem vettem figyelembe a multiplatform működéshez szükséges dolgokat, így az elkészült alkalmazás csak webalkalmazásként és Android alkalmazásként működik. Habár ezek megjelenésbeli hasonlósága lenyűgöző, működésükben még mindig különbözőek, így oda kell figyelni fejlesztéskor, hogy a különböző platformokon megfelelően működjenek az alkalmazások. Véleményem szerint az XAML nyelvű felületleírás, ha bár hosszabb és néha átláthatatlanabb kódot eredményez, de összeségében alkalmasabb lehet bonyolult felületek létrehozására, mint a html.



Ha kizárólag webalkalmazás és nem multiplatform fejlesztés a cél WebAssembly felett, akkor az Uno Platform jelenlegi állapotában még elmarad a Blazor képességei mellett, valamint multiplatform fejlesztés esetén is meggondolandó egy Blazor-ös Progressive Web Application fejlesztése, mivel ahhoz, hogy az Uno-val fejlesztett alkalmazás minden általa támogatott platformon megfelelően működjön sok, a hatékony fejlesztést hátráltató megkötést kell betartani.

A teljes forráskód elérhetősége: https://github.com/mzsb/Wiki