LAMK
Lahden ammattikorkeakoulu
Lahti University of Applied Sciences

TRAVELING BLAZOR WEBASSEMBLY

WITH SIMPLE FOOD RECIPE PAGE

Lahti University of Applied Science
Bachelor of Engineering
Software Engineering
Spring 2020
Lumi Kwon

SISÄLLYS

1 INTRODUCTION

Web application is cross-platform choice, which works both on Desktop operating systems and on mobile systems. Web application consists of web pages, and web page has two sides: back-end, which works on server-side and front-end, which works on client-side (Lvivity 2018). Unlike back-end programming, front-end programming language is limited. NPAPI (Netscape Plugin Application Programming Interface)-based applications, such as Adobe Flash, Java Applet and ActiveX, forces users to install some programs to use the application. Nowadays they are considered as harmful technology, due to the chance of malicious code inside the installation program. Due to the problem, Firefox already stopped the NPAPI support since 2017 (Mozilla 2020). Without any front-end program that forces user to install application, Javascript is the only choice as the standard language that works on webpages.

But Javascript work differently than other languages, which makes it difficult to learn. One of its unusual features is *undefined*, which sometimes acts as *null*, even the *null* exists as an object. If unassigned variable is read in Javascript, it doesn’t throw any null exception, but it returns undefined, which can be changed to other thing. For example, string that contains literally “undefined” by adding to string or turns number into NaN by adding to number. Because of this, some languages are introduced, which can be transpiled into Javascript, such as Typescript. However, such languages basically result Javascript, which means the feature of Javascript cannot be fully avoided. That’s why Typescript is sometimes criticized (Chen 2019). Another problem of Javascript is its performance. Even Javascript itself is not slow language, still performance is limited, since it’s non-strict typed and interpreted language. WebAssembly and Blazor can be solution of these problems.

The research objectives are to study what is Blazor, especially what is Blazor WebAssembly and how to use it. The example application is simple recipe web page, which finds recipe by ingredients as keywords. The result gives all recipe that contains all the ingredients. When clicking the recipe name, it shows recipe page. All processes retrieve JSON API data from the server, rather than requiring the whole HTML page, which means, the page is SPA (Single Page Application). At server side, ASP.NET Core API and SQL Server is used to execute locally. However, testing and evaluating will be only focused on client-side Blazor. Keyword search box is reusable component in this project. The testing targets both component and whole client-side Blazor app.

2 WEBASSEMBLY

WebAssembly is used to make front-end logic without any Javascript code. WebAssembly can perform at nearly the speed of code compiled for a native platform (Mercier 2019). WebAssembly targets basically C, C++ and Rust (Mercier 2019), and it is also currently available in many other languages, such as Java and C# (Akinyemi 2019).

WebAssembly is a standardized form of asm.js (WebAssembly 2020a). Asm.js is a subset of Javascript. Asm.js eliminated dynamic typing and garbage collection to increase performance, which tries to be closer to C or C++ than Javascript. Also asm.js is compiled before execution (AOT, Ahead Of Time), which results predictable performance. (Asm.js 2020) At first the focus was on compiling C and C++ into WebAssembly, but currently high-level goal is to improve support for other languages. (WebAssembly 2020a).

WebAssembly is not replacement for Javascript, but it is rather complemental with Javascript. It is possible to use WebAssembly with Javascript. (WebAssembly 2020a.) For example, UI communication can be done with Javascript and logical side can be made with WebAssembly. In this case, WebAssembly can be reusable module for complicated application. (WebAssembly 2020b.)

Despite the major usage is currently front-end, WebAssembly is not only aimed for front-end development. WebAssembly can be used outside the web browser thanks to WebAssembly System Interface (WASI). WASI makes it possible to create Server-side app. (Eberhardt 2019.)

3 ASP.NET CORE

ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-based, internet-connected applications. ASP.NET can be used for Web Apps, IoT Apps and Mobile backends. ASP.NET works on server-side, except Blazor WebAssembly. (Roth D. & Anderson R. & Luttin S. 2019.)

ASP.NET is architected for testability, and it natively supports dependency injection, which implements the idea of dependency inversion principle: a class should depend on an interface, not an implementation. (Roth D. et al. 2019.)

MVC (Model-View-Controller) is common pattern for ASP.NET Core, but other choices are available too, such as Razor Page based solution. In Razor Page based solution, routing is defined each Razor Pages, not controllers. (Roth D. et al. 2019.)

ASP.NET has Startup.cs, which configurates services and the app's request pipeline. In Startup.cs, some services are defined such as configuration for dependency injection and routing. It's also possible to add configuration only for development, for example, throwing developer exception page when error occurs. (Anderson R. & Dykstra T. & Latham L. & Smith S. 2019.)

4 BLAZOR

Blazor is a framework building interactive client-side web UI with .NET (Roth D. & Latham L. 2020a). Currently there are two Blazor types available – Blazor Server and Blazor WebAssembly. The left picture is how Blazor Server works and right picture is how Blazor WebAssembly works (Figure 1). Both does not require front-end developers to use Javascript (Roth 2019), but there are Javascript interop, which gives possibility to use Javascript in Blazor.

Blazor Server communicates to server with real-time SignalR connection. SignalR is open-source library that simplifies adding real-time web functionality apps. SignalR pushes the content from server to the client, which fits for scenario that requires high frequency updates. (Microsoft 2019a.) Since the process requires network connection, offline scenario for Blazor Server is not supported (Roth 2019).

Blazor Webassembly works fully on client-side by using WebAssembly-based .NET runtime. Unlike Blazor Server, in the Blazor WebAssembly, the UI events are handled directly in the client side. Since it is basically bunch of static files, it can be used for any static website hosting solution, GitHub site, for example. (Roth 2020a.) Blazor WebAssembly is currently in preview and planned to ship in May 2020 (Roth 2020a). In other words, Blazor WebAssembly is not yet ready for production.

With Blazor WebAssembly, ASP.NET back-end developer can learn Blazor fast and develop the full-stack level, both front-end and back-end. Developing both side with one technology requires less learning curve, also developer can focus on and understand one technology deeper. There is no .NET server-side dependency, thus developer can choose backend language or platform with their own flavour – Blazor WebAssembly is just front-end framework and works well whatever server-side backend is. (Roth 2020b.)

Since Blazor WebAssembly is inspired by existing SPA (Single Page Application) frameworks (DotNet 2019), it natively supports SPA.

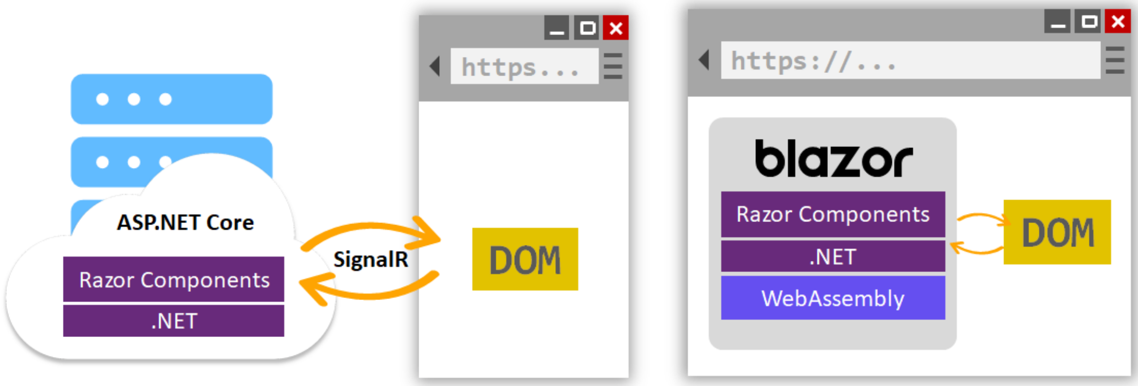


Figure 1. Blazor Server and Blazor WebAssembly (Microsoft 2020a)

4.1 Structure

When app is started, Blazor loads bunch of DLL files. The DLL files are components and libraries added by system and user. Blazor also loads blazor.webassemly.js, mono.wasm and mono.js. Blazor.webassembly.js loads the .NET runtime, the app, and the app's dependencies (Roth 2020b). Mono.wasm is compiled WebAssembly from Mono Framework, which contains actual Mono WebAssembly for .NET runtime. Mono.js loads the WebAssembly module and passes it to Blazor application assembly. Mono.js also contains Javascript Interop for Blazor. (Glick 2018.)

Mono aims to run under WebAssembly in two modes: interpreted and AOT (Figure 2). In the left side of figure is interpreted mode. In interpreted mode, Mono runtime is compiled but .NET assembly files are not compiled. In development mode, interpreted mode is much faster than AOT, because it saves compilation time. AOT mode, the right side of figure, uses pure WebAssembly binaries that are already compiled. AOT takes some time to compile, but since it's already compiled, it's faster than interpret mode. (Sanderson 2018a.)

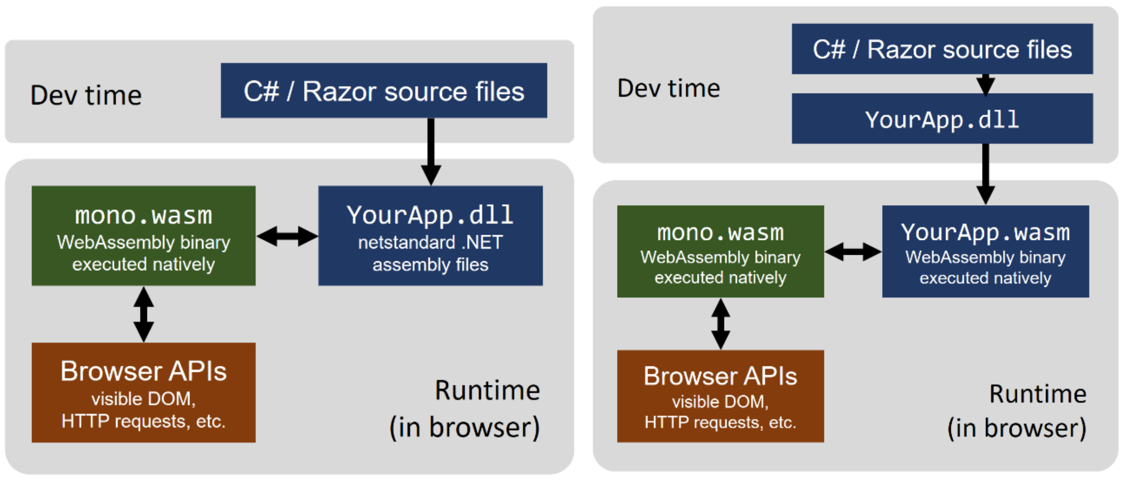

Dev time
C# / Razor source files
mono.wasm
WebAssembly binary executed natively
YourApp.dll
netstandard .NET assembly files
Browser APIs
visible DOM, HTTP requests, etc.
Runtime (in browser)
Dev time
C# / Razor source files
YourApp.dll
mono.wasm
WebAssembly binary executed natively
YourApp.wasm
WebAssembly binary executed natively
Browser APIs
visible DOM, HTTP requests, etc.
Runtime (in browser)

Figure 2. Interpreted mode and AOT mode of Blazor (Sanderson 2018a)

Blazor is part of the ASP.NET project, which means, Blazor code template are basically very similar to ASP.NET code. Like ASP.NET, Blazor can add services using Startup file or directly in main program. Blazor also has Razor pages, which contains HTML, routing path and C# code.

The final web root, which does not contain C# codes, resides wwwroot directory, as index.html. Index.html contains HTML head, loading page, default CSS and Javascript calls. Wwwroot also contains CSS and JavaScript files, which doesn't need to be compiled.

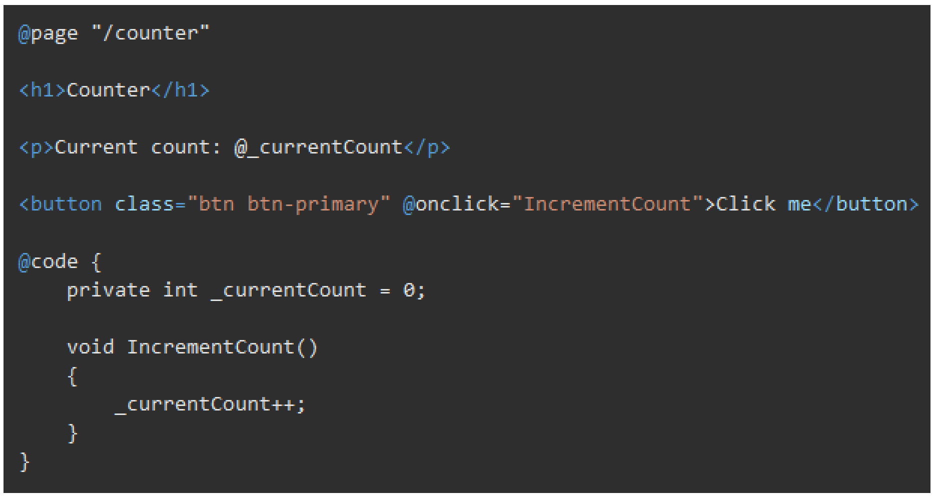
4.2 Retrieving data

Although Blazor default client template provides automatically ASP.NET API on server-side, Blazor works any API without the server-side project. In other words, Blazor WebAssembly does not have server dependency. (Roth 2020b) Blazor provides REST API methods (Get, Post, Put, Delete) as default. Default REST API can also parse data to JSON via *GetJsonAsync()* and *PostJsonAsync()*.

As another option, gRPC is available for Blazor, which is a high-performance of RPC (Remote Procedure Call) developed by Google. gRPC is optimized for minimal network traffic, which fits with SPA. (Sanderson 2020.)

4.3 Razor

Blazor uses Razor file for every pages, components and layouts (Roth D. et al. 2020a). Razor is page-focused framework for building dynamic, data-driven web sites with clean separation of concerns. Razor page contains both HTML and C# code for ease to use, as well as page routing definition for using GET URI parameters. Razor was already used in server-side application before Blazor is created. (Learn Razor Pages 2019.) Razor file can be external class library for reusable component. *_Imports.razor* contains a global *using directive* for every Razor file in the assembly.



```
@page "/counter"

<h1>Counter</h1>

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

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

@code {
    private int _currentCount = 0;

    void IncrementCount()
    {
        _currentCount++;
    }
}
```

Figure 3. Syntax of Blazor (Latham L. et al. 2020a)

Razor page can have several directives on the top of the code (Figure 3). The *@page* directive indicates the routing endpoint. Without *@page* directive, user cannot directly access the page via URI, but it’s still possible to use it as component in other Razor code. *@code* directive have some C# class members. (Roth D. & Latham L. 2020b.) It can contain methods, which is triggered when the Razor component is loaded or rendered, or parameter is set. For example, *OnInitializedAsync()* is called when page is loaded. (Latham L. & Roth D. 2019.)

4.3.1 Directives

Razor page can get parameter by passing value in the component or by passing into the URI. Route parameter is defined on *@page* directive, inside *{variable name}*. Component parameter is passed through markup attribute when adding the component. Both get the variable by adding *parameter* attribute on the public property. (Latham L. & Roth D. 2020a.)

Layout folder contains layouts. Default layout is MainLayout.razor. Layout is defined by inheriting from layout component base by declaring *@inherits LayoutComponentBase* on the top of the Razor content and *@Body* is inserted in the code where the content should fit. In the page, the layout can be applied using *@layout LayoutName*. In this way, it’s possible to make nested layout (Rainer S. & Latham L. 2019).

4.3.2 Dependency Injection

Since Blazor is part of the ASP.NET, pure C# file (.cs) can be separated and the class can be created directly or by injecting it in the Razor file. When injecting the code, the C# class is declared in the ASP.NET configuration and added to Razor file with *@inject*. Blazor can use Startup.cs file to register the dependency injection or the injection can be directly registered inside Program.cs.

Some of Dependency Injection is enabled by default without registering to services: *HttpClient*, *IJSRuntime*, *NavigationManager*. *HttpClient* is used for calling Web API, *IJSRuntime* is used for Javascript Interop and *NavigationManager* is used for navigating URI. (Rainer 2020.)

4.3.3 Data Bindings

On Razor element tree, parent data can be bound with their child and, in some case, vice versa. The first case is one-way binding and the second case is two-way binding.

When values can be updated only from parent, the data flow is unidirectional. This is one-way data binding. One-way data binding can be realized by passing value as parameter to the child. (Sainty 2019.)

When the value is updated from two directions, the data flow is bidirectional, thus it’s two-way data binding. Two-way data binding is realized with *@bind* or *@bind-{parameter}* directive. For example, if *@bind* is used on text input, the data is updated when the text input value is changed. (Sainty 2019.)

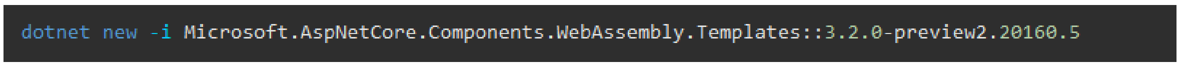
@bind-{parameter} is used between parent and child component. Event to bind can be specified using *@bind-{parameter}:{event}* directive. When child element is created with two-way bindings, *EventCallBack<T>* should be specified and invoked to implement two-way binding. (Sainty 2019.)

5 IMPLEMENTATION

The recipe app searches recipe by ingredient name, not by food name. The idea of recipe app is finding recipes that can be made from leftover ingredients at home. Recipe ingredient is keyword, which means, the input is limited to data from database ingredients. The result shows food recipes that contains any ingredient from list.

On client-side, Blazor WebAssembly is used. All source codes are written in C# and Razor Syntax, without any Javascript interop. On server-side, ASP.NET with SQL Server file is used for database logic. Client assembly and Server-side assembly is separated.

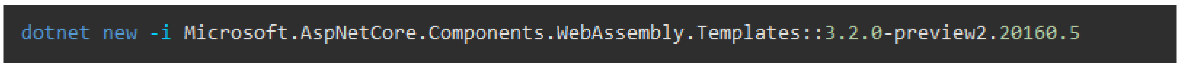
5.1 Creating Application



```
dotnet new -i Microsoft.AspNetCore.Components.WebAssembly.Templates::3.2.0-preview2.20160.5
```

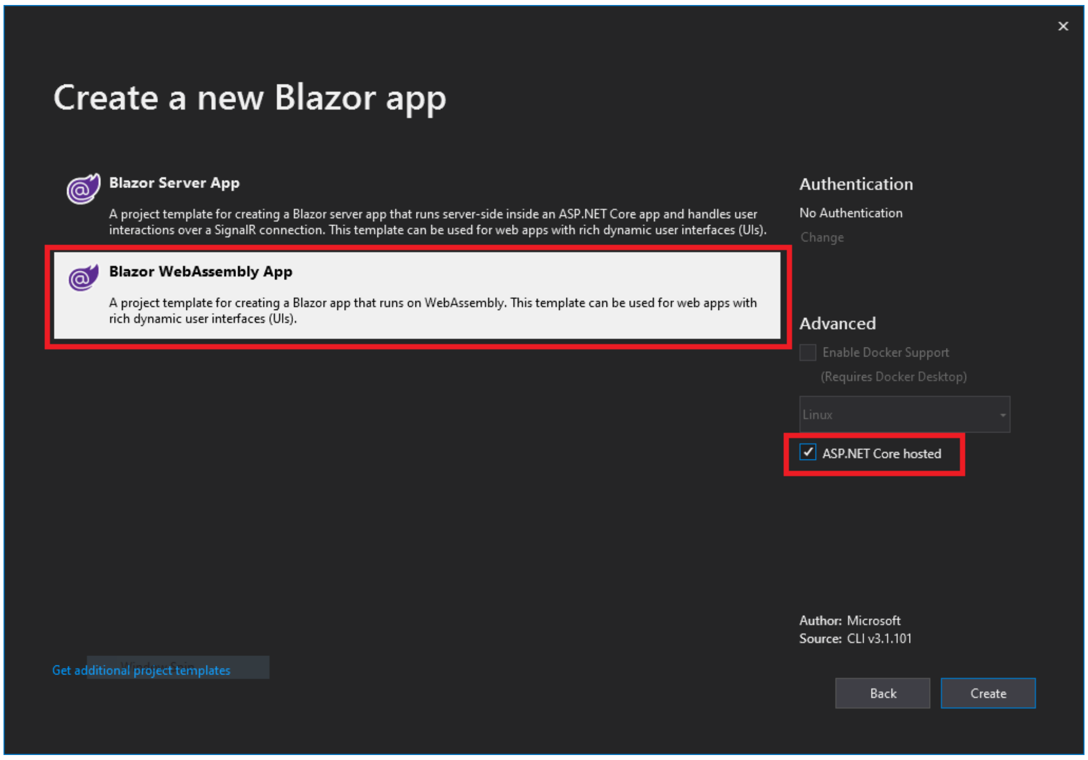
Figure 4. Enabling Blazor WebAssembly on .NET Core CLI (Roth D. & Latham L. 2019.)

Blazor WebAssembly preview requires ASP.NET Core 3.1. Blazor WebAssembly has example template, which can be edited for own use. This template is available from .NET Core Command line (



```
dotnet new -i Microsoft.AspNetCore.Components.WebAssembly.Templates::3.2.0-preview2.20160.5
```

Figure 4). (Roth D. & Latham L. 2019.) After the command, when the Blazor WebAssembly App is shown on the template list in the command output, the template is installed.


Create a new Blazor app

Blazor Server App
A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).

Blazor WebAssembly App
A project template for creating a Blazor app that runs on WebAssembly. This template can be used for web apps with rich dynamic user interfaces (UIs).

Authentication
No Authentication
Change

Advanced
Enable Docker Support
(Requires Docker Desktop)
Linux
[✓ ASP.NET Core hosted]

Author: Microsoft
Source: CLI v3.1.101

Figure 5. Blazor app creation dialog

After the command is executed, Visual Studio shows *Blazor App* when creating a new project. After creating a new project, it shows dialog for asking which one will be created, Blazor Server App or Blazor WebAssembly App (Figure 5). ASP.NET Core hosted app creates automatically server-side API project.

5.2 KeywordSearchBox class library

For researching how external component works, search box and result box component is outside of the Blazor WebAssembly main assembly. The KeywordSearchBox Class is Razor Class Library.

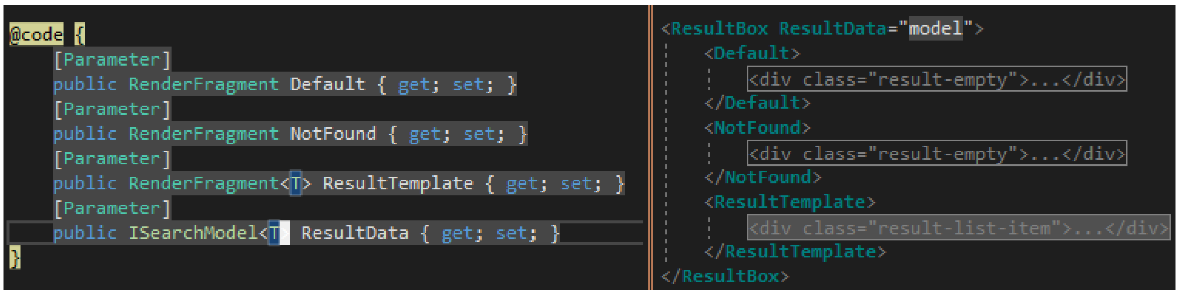
KeywordSearchBox class library consists of two components – Searchbox, which provides search bar and Resultbox, which shows result. The two components are separated, so user can decide where to place the results. For example, user can place results inside div element, or toggle it to show or hide.

The assembly contains Razor pages, logics, and models. Logics defines what to do with DOM event, such as searching or keyboard input actions. Models contains list of words, such as current words or suggested words from input value. There are also public interfaces and classes available: ISearchModel<T> and SearchModel<T>. ISearchModel<T> contains keywords as Enumerable string and results as Enumerable generic type.

KeywordSearchBox provides only common logics for filling keywords and binding results. User defines search action, type and result template.

@typeparam directive declares generic type for a Razor page. The type is used by parameter type and it's determined by passing parameter. (Latham L. & Roth D. 2020b.) In this project, ISearchModel<T> parameter uses generic type.

The Razor component can get user-defined HTML tag by getting *RenderFragment* as parameter. The parameter name *ChildContent* as *RenderFragment* indicates direct content of the component. It's possible to define custom parameters by defining the parameter name of *RenderFragment*. Then, the search box library users can define each template using XML syntax inside component when component is consumed. (Figure 6) (Charbeneau 2020.)



```
@code {
    [Parameter]
    public RenderFragment Default { get; set; }
    [Parameter]
    public RenderFragment NotFound { get; set; }
    [Parameter]
    public RenderFragment<T> ResultTemplate { get; set; }
    [Parameter]
    public ISearchModel<T> ResultData { get; set; }
}
```

```
<ResultBox ResultData="model">
    <Default>
        <div class="result-empty">...</div>
    </Default>
    <NotFound>
        <div class="result-empty">...</div>
    </NotFound>
    <ResultTemplate>
        <div class="result-list-item">...</div>
    </ResultTemplate>
</ResultBox>
```

Figure 6. Example of *RenderFragment* usage

With the *RenderFragment*, users can choose where to put the model data via *context*. The default variable name is *context*, but it's possible to define context name with *context="variableName"* (Latham L. & Roth D. 2020b). *RenderFragment* should be generic type for this feature to know what the *context* is.

On the result page, the generic *RenderFragment* is used to let user to define the template using model data. In this case, users don't need to write *foreach* loop in their code to show the list of results.

5.3 Blazor WebAssembly Main

The recipe application has two pages and a component. The main page shows search box and the search results. Another page shows recipe by Recipe ID from database. The component is search box, which contains search box and result box from KeywordSearchBox class library. The project has only one Layout, which contains logo.

Search component defines action, what will do when user press search button. It retrieves data from server using *GetJsonAsync()* from injected *HttpClient*. There may be multiple keywords and the keywords can be long, so retrieving by passing the keywords in the URL via GET method can be nasty. Thus, it uses POST method to send request, by passing keyword array as body.

The Search box has dependency on the search box Razor page. ISearchModel<T> is added to startup services as Singleton and the service is injected to the Razor page. The injected ISearchModel<T> is two-way bound with search box component. The binding shows result instantly when user clicks the search button.

5.3.1 Adding Dependency

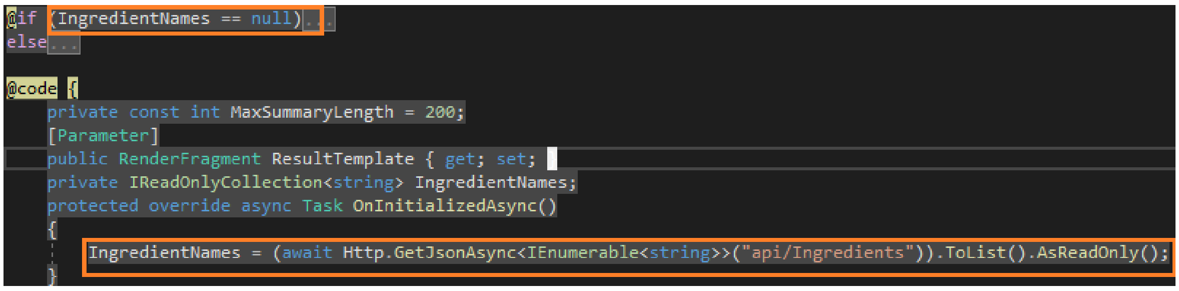
To contain a Razor component in the project in Visual Studio, right click the *dependences* and select *add references*. If the project is in the same solution, the project can be added from *projects* tab. If the project is already compiled file, such as EXE or DLL file, the reference can be added through clicking *Browse…* on the bottom of window and select the reference file. This is not applied for only Razor class libraries, but also for other external references. (Microsoft 2019b.)

Since Razor class library doesn't contain main page, main page should add externally web resources, such as CSS or JavaScript. In this project, the main page is *wwwroot/index.html* in the Client project. The CSS or Javascript from the class library can be added by just adding HTML code that calls CSS or JavaScript in *wwwroot/index.html*. The URL of resource is *_content/Project Name/File path*. The *File path* is relative path from *wwwroot*, from the Razor class library. (Microsoft 2020a.) In this project, the CSS is linked from *_content/KeywordSearchBox/styles.css*.

5.3.2 Loading pages

Whole loading page can be set within the *app* block in the *wwwroot/index.html*. When the page is loaded, the loading page inside *app* block is replaced to content.

Blazor needs render placeholder UI elements when the page is loaded but data is not yet fetched asynchronously. To implement this, the variable is defined without assigning at first. The variable will be *null* by default, when the variable is class. Then data is fetched on OnInitializedAsync() block, which is called when the page is loaded. Finally, loading elements can be shown by using *if-else* blocks on the Razor page. If the data is null, the page shows loading elements, else the page shows the data. The *if-else* block checks if the data is null, so it shows data when data is loaded. (Figure 7.) (Latham L. et al. 2019.)



```
@if (IngredientNames == null)...
else...

@code {
    private const int MaxSummaryLength = 200;
    [Parameter]
    public RenderFragment ResultTemplate { get; set; }
    private IReadOnlyCollection<string> IngredientNames;
    protected override async Task OnInitializedAsync()
    {
        IngredientNames = (await Http.GetJsonAsync<IEnumerable<string>>("api/Ingredients")).ToList().AsReadOnly();
    }
```

Figure 7. Handling with loading page for asynchronously fetched data

5.3.3 Handling fetch error

GetJsonAsync and *PostJsonAsync* use *SendJsonAsync*. In the *SendJsonAsync*, when the server returns erroneous result, the program throws *HttpRequestException* using *EnsureSuccessStatusCode()*. (Sanderson 2018b.) The *EnsureSuccessStatusCode* throws exception when *IsSuccessCode* is false, which checks if the response code is range of 200-299 (Microsoft 2020b).

In other words, the client can catch error from server by catching *HttpRequestException*. However, with *SendJsonAsync*, there is still no way to check which error code server returned, without parsing the exception message (Sanderson 2018b; Microsoft 2020c). In the application, the error message is just showing something error is happened.

There is more thing to consider, when use *SendJsonAsync* with handling error with ASP.NET Core API. The server API returns *204 No content*, when data is null (Strahl 2020.). This happened when the recipe ID was wrong in the application. The database returned empty dataset when recipe ID is not correct. It caused server API to return null, and client received 204 instead of error code. 204 is still inside 200-299 range, so the *SendJsonAsync* doesn’t throw *HttpRequestException*. But in the parsing phase, the empty string is invalid JSON, and it throws *JsonException*. There are some ways to solve it: Client catches JsonException, or server sends erroneous code when the data is null. In this case, when data is not found from database, throwing 404 is reasonable, even from the server-side API. Thus, the application is used the server-side solution.

6 TESTING

Since the goal of work is researching Blazor WebAssembly, testing has done with only client-side. The tool for testing is XUnit and Bunit. XUnit is one of the popular unit testing tool for C# and BUnit is testing tool for Blazor.

Razor component instance for testing can be created with *new ComponentName()*, but using it this way have limitations. First, it cannot trigger event handlers or invoke renderers. Second, it doesn't verify event callbacks or rendered markups. (Hansen 2020a.) There is no way to find element while page is loading.

However, the service-level unit testing can be covered with any C# testing framework, since it doesn't contain any HTML-like component. For this purpose, XUnit is used for testing methods in C# files.

The KeywordSearchBox is used as a library, so the internal methods should not be exposed to public. But the internal methods should be visible for testing assembly.

6.1 C# file unit tests

The attribute *assembly:System.Runtime.CompilerService.InternalsVisibleTo* exposes internal types in current assembly to specific assembly. Public key can be specified in the attribute option. In this case, the path of *keyfile* is given as compiler parameter. Keyfile is generated by strong name tool. (Microsoft 2020d.) Strong name is consists of assembly's identity, which also contains digital signature. However, all of .NET Core assemblies are signed, so this project doesn't need to assign the key. (Microsoft 2020e.)

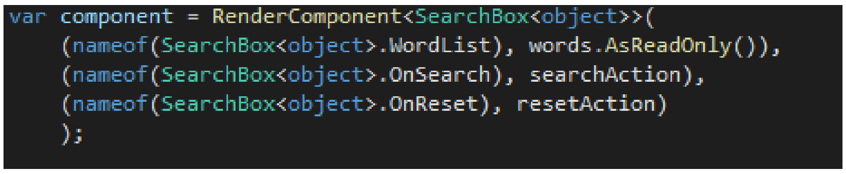
The *InternalsVisibleTo* is used by putting once any C# file in the current assembly. The attribute lay on a namespace.

XUnit uses *Fact* attribute for testing without data input, and *Theory* attribute for testing with multiple data input. Theory data can be defined directly in the *InlineData* attribute or defined in specific class or method. In this case, the class implements *IEnumerable<object[]>* and the method returns *IEnumerable<object[]>*. (Lock 2017.) The *Fact* and *Theory* attributes are on each method and the method should be public for testing. In the project, only *Fact* attribute is used.

6.2 Razor component tests

With BUnit, not only Razor unit tests can be performed but black-box test can be done. In this project, BUnit is used for integration test.

BUnit provides C#-based testing, Razor-based testing and snapshot-based testing. C#-based testing uses pure C# with XUnit. It is executed just like regular C# tests. Razor-based testing writes components for testing in Razor page. It allows to declare components in Razor syntax. The test assertion is still written in C# in the *@code* block, though. Snapshot test uses only markup for writing testing and comparison. It uses very little C# code, usually for configuring services. Razor-based tests and snapshot-based tests are experimental feature and the syntax and API will be likely changed. (Hansen 2020b.) The project doesn't contain complicated component that needs many parameters, so only C#-based testing did the work.



```
var component = RenderComponent<SearchBox<object>>(
    (nameof(SearchBox<object>.WordList), words.AsReadOnly()),
    (nameof(SearchBox<object>.OnSearch), searchAction),
    (nameof(SearchBox<object>.OnReset), resetAction)
    );
```

Figure 8. Adding component for testing

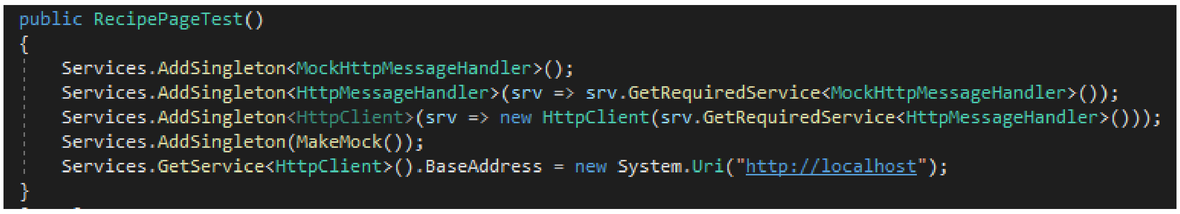
The testing target class inherits from *ComponentTextFixture* for C#-based testing. Then, the component is rendered by using *RenderComponent<ComponentName>()*. The component parameters can be given by passing parameters to *RenderComponent*. *RenderComponent<T>* returns *IRenderedComponent<T>*. (Figure 8.) (Hansen 2020c.)

Next, the element from component for testing is acquired. *IRenderedComponent.Find(CssSelector)* selects the desired element from the component (Hansen 2020c). It returns *AngleSharp.Dom.IElement*. AngleSharp is popular angle brackets parser library for C#, including HTML5 and CSS support. Once it returns AngleSharp DOM element, it can be used such as adding event or investigating the result. For example, input or clicking can be done, child or parent element can be found, it or just the text inside can be read or written.

When *Egil* testing framework is changed to *Bunit*, it changed to apply automatically when the component is changed (Hansen 2020d). Before then, *IRenderedComponent.Find* was used after each input for suggestion testing. The excessive *Find* methods are removed, which gets same element as before.

Element's non-existence assertion was not supported directly from library. For assure that specific element does not exist, *Assert.Throws<ElementNotFoundException>()* is used. This way used that the the *IRenderedComponent.Find* method throws *ElementNotFoundException* when the method didn't find the element. The exception is also part of the Bunit.

BUnit removed *AddMockHttp()* for removing third-party dependency. For this reason, HTTP mocking is done manually. First, the mocked HTTP message handler service is added. Then, HTTP message handler service from the mocked service is added. Finally, HTTP client from the message handler is added. (Hansen 2020a.) Base address of HTTP client should be defined using *GetService().BaseAddress* when the Razor page for testing uses relative URI. (Figure 9.) The mocking should be performed before rendering component, or the testing doesn't work.



```
public RecipePageTest()
{
    Services.AddSingleton<MockHttpMessageHandler>();
    Services.AddSingleton<HttpMessageHandler>(srv => srv.GetRequiredService<MockHttpMessageHandler>());
    Services.AddSingleton<HttpClient>(srv => new HttpClient(srv.GetRequiredService<HttpMessageHandler>()));
    Services.AddSingleton(MakeMock());
    Services.GetService<HttpClient>().BaseAddress = new System.Uri("http://localhost");
}
```

Figure 9. Mocking HTTP client for testing

7 CONCLUSION

The whole programming process, including implementation and testing, was planned for one month. It was not so strict plan; however, program went through minor edits after the month. The change was done for applying better solution from new knowledge.

During the writing process, Blazor was still evolving. Blazor already released preview 2. Thanks for the community, Blazor WebAssembly is well-polished and soon it’s ready for product. Especially the BUnit testing framework changed notably. It caused to install the new testing package and to edit the testing codes.

The goal of the project itself was getting familiar to Blazor, and no further development will be done. However, the knowledge will be applied on other Blazor softwares, including my portfolio. When the Blazor is ready for products, it can be used commercial purpose too.

REFERENCES

Akinyemi S. 2019. Awesome WebAssembly Languages. Github [accessed 30.1.2020]. Saatavissa: https://github.com/appcypher/awesome-wasm-langs

Anderson R., Dykstra T., Latham L., Smith S. 2019. App startup in ASP.NET Core. Microsoft Docs [accessed 18.2.2020]. Available at: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/startup?view=aspnetcore-3.1

Asm.js 2020. asm.js – frequently asked questions [accessed 8.2.2020]. Available at: http://asmjs.org/faq.html

Charbeneau E. 2020. Authoring Custom Components. Video [accessed 8.3.2020]. Available at: https://www.youtube.com/watch?v=sCrQpYL2vyg&list=PLdo4fOcmZ0oWlP1Qpzg7Dwzxr298ewdUQ&index=10&t=0s

Chen L. 2019. TypeScript is a waste of time. Change my mind. Dev Community [accessed 7.2.2020]. Available at: https://dev.to/bettercodingacademy/typescript-is-a-waste-of-time-change-my-mind-pi8

DotNet 2020. FAQ dotnet/blazor Wiki. Github [accessed 30.1.2020]. Available at: https://github.com/dotnet/blazor/wiki/FAQ

Eberhardt C. 2019. WebAssembly 2019 Year In Review. Scott Logic [accessed 7.2.2020]. Available at: https://blog.scottlogic.com/2019/12/24/webassembly-2019.html

Glick D. Blazor, Razor, WebAssembly, and Mono. Blog [accessed 8.2.2020]. Available at: https://daveaglick.com/posts/blazor-razor-webassembly-and-mono

Hansen E. 2020a. Remove MockHttp logic from library #56. Github [accessed 14.3.2020]. Available at: https://github.com/egil/bunit/issues/56

Hansen E. 2020b. Basics of Blazor component testing. Documentation [accessed 15.3.2020]. Available at: https://bunit.egilhansen.com/docs/basics-of-blazor-component-testing.html

Hansen E. 2020c. C# Test Examples. Documentation [accessed 15.3.2020]. Available at: https://bunit.egilhansen.com/docs/csharp-test-examples.html

Hansen E. 2020d. Twitter [accessed 15.3.2020]. Available at: https://twitter.com/egilhansen/status/1234057667007713281?s=20

Latham L., Roth D. 2019. ASP.NET Core Blazor lifecycle. Microsoft Docs [accessed 18.2.2020]. Available at: https://docs.microsoft.com/en-us/aspnet/core/blazor/lifecycle?view=aspnetcore-3.1

Latham L., Roth D. 2020a. Create and use ASP.NET Core Razor Components. Microsoft Docs [accessed 18.2.2020]. Available at: https://docs.microsoft.com/en-us/aspnet/core/blazor/components?view=aspnetcore-3.1

Latham L., Roth D. 2020b. ASP.NET Core Blazor templated components. Microsoft Docs [accessed 8.3.2020]. Available at: https://docs.microsoft.com/en-us/aspnet/core/blazor/templated-components?view=aspnetcore-3.1

Learn Razor Pages 2019 [accessed 9.2.2020]. Available at: https://www.learnrazorpages.com/

Lock A. 2017. Creating parameterised tests in xUnit with [InlineData], [ClassData], and [MemberData]. Blog [accessed 14.3.2020]. Available at: https://andrewlock.net/creating-parameterised-tests-in-xunit-with-inlinedata-classdata-and-memberdata/

Lvivity 2018. Front-end and back-end. Interaction in plain words [accessed 7.2.2020]. Available at: https://lvivity.com/front-end-back-end-interaction

Mercier, C. 2019. W3C RECOMMENDS WEBASSEMBLY TO PUSH THE LIMITS FOR SPEED, EFFICIENCY AND RESPONSIVENESS. W3C [accessed 30.1.2020]. Available at: https://www.w3.org/blog/news/archives/8123

Microsoft 2019a. Introduction to ASP.NET Core SignalR. Microsoft Docs [accessed 31.1.2020]. Available at: https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-3.1

Microsoft 2019b. Manage references in a project. Microsoft Docs [accessed 11.3.2020]. Available at: https://docs.microsoft.com/en-us/visualstudio/ide/managing-references-in-a-project?view=vs-2019

Microsoft 2020a. Create reusable UI using the Razor class library project in ASP.NET Core. Microsoft Docs [accessed 11.3.2020]. Available at: https://docs.microsoft.com/en-us/aspnet/core/razor-pages/ui-class?view=aspnetcore-3.1&tabs=visual-studio

Microsoft 2020b. HttpResponseMessage.EnsureSuccessStatusCode Method. Microsoft Docs [accessed 13.3.2020]. Available at: https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpresponsemessage.ensuresuccessstatuscode?view=netframework-4.8

Microsoft 2020c. HttpRequestException Class. Microsoft Docs [accessed 13.3.2020]. Available at: https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httprequestexception?view=netframework-4.8

Microsoft 2020d. InternalsVisibleToAttribute Class. Microsoft Docs [accessed 14.3.2020]. Available at: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.compilerservices.internalsvisibletoattribute?view=netframework-4.8

Microsoft 2020e. Create and use strong-named assemblies. Microsoft Docs [accessed 14.3.2020]. Available at: https://docs.microsoft.com/en-us/dotnet/standard/assembly/create-use-strong-named

Mozilla 2020. Why do Java, Silverlight, Adobe Acrobat and other plugins no longer work? [accessed 7.2.2020]. Available at: https://support.mozilla.org/en-US/kb/npapi-plugins

Rainer S. 2020. ASP.NET Core Blazor dependency injection. Microsoft Docs [accessed 8.2.2020]. Available at: https://docs.microsoft.com/en-us/aspnet/core/blazor/dependency-injection?view=aspnetcore-3.1

Rainer S., Latham L. 2019. ASP.NET Core Blazor layouts. Microsoft Docs [accessed 9.2.2020]. Available at: https://docs.microsoft.com/en-us/aspnet/core/blazor/layouts?view=aspnetcore-3.1

Roth D. 2020a. Welcome To Blazor. Video [accessed 8.2.2020]. https://www.youtube.com/watch?v=KlngrOF6RPw&list=PLdo4fOcmZ0oWlP1Qpzg7Dwzxr298ewdUQ&index=2&t=0s

Roth D. 2020b. ASP.NET Core Blazor Hosting Models. Microsoft Docs [accessed 8.2.2020]. Available at: https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-3.1

Roth D., Anderson R., Luttin S. 2019. Introduction to ASP.NET Core. Microsoft Docs [accessed 15.2.2020]. Available at: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-3.1

Roth D., Latham L. 2019. Get Started with ASP.NET Core Blazor. Microsoft Docs [accessed 7.3.2020]. Available at: https://docs.microsoft.com/en-us/aspnet/core/blazor/get-started?view=aspnetcore-3.1&tabs=visual-studio

Roth D., Latham L. 2020a. Introduction to ASP.NET Core Blazor. Microsoft Docs [accessed 8.2.2020]. Available at: https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-3.1

Roth D., Latham L. 2020b. Build your first Blazor app. Microsoft Docs [accessed 20.2.2020]. Available at: https://docs.microsoft.com/en-us/aspnet/core/tutorials/build-your-first-blazor-app?view=aspnetcore-3.1

Roth, D. 2019. Blazor Server in .NET Core 3.0 scenarios and performance. Microsoft ASP.NET Blog [accessed 31.1.2020]. Available at: https://devblogs.microsoft.com/aspnet/blazor-server-in-net-core-3-0-scenarios-and-performance/

Sanderson S. 2018a. Blazor: A technical introduction. Blog [accessed 8.2.2020]. Available at: http://blog.stevensanderson.com/2018/02/06/blazor-intro/

Sanderson S. 2018b. Add Extension Methods for HTTP JSON requests/responses. Github [accessed 13.3.2020]. Available at: https://github.com/dotnet/aspnetcore/commit/a74124efbf734681457812a0d5360ef5cbcf53b4

Sanderson S. 2020. Using gRPC-Web with Blazor WebAssembly. Blog [accessed 8.2.2020]. Available at: https://blog.stevensanderson.com/2020/01/15/2020-01-15-grpc-web-in-blazor-webassembly/

Sainty C. 2019. A Detailed Look At Data Binding in Blazor. Blog [accessed 6.3.2020]. Available at: https://chrissainty.com/a-detailed-look-at-data-binding-in-blazor/

Strahl R. 2020. Null API Responses and HTTP 204 Results in ASP.NET Core. Blog [accessed 13.3.2020]. Available at: https://weblog.west-wind.com/posts/2020/Feb/24/Null-API-Responses-and-HTTP-204-Results-in-ASPNET-Core

WebAssembly 2020a. FAQ [accessed 8.2.2020]. Available at: https://webassembly.org/docs/faq/

WebAssembly 2020b. Use Cases [accessed 8.2.2020]. Available at: https://webassembly.org/docs/use-cases/

LIITTEET