Sisällys

[Blazor WASM demo, eli erillisen API:n päälle rakennettu frontti 1](#_Toc73114711)

[Mikä: 1](#_Toc73114712)

[Miksi: 1](#_Toc73114713)

[Kälileiskat: 1](#_Toc73114714)

[Datan välitys palvelimen ja frontin välillä 2](#_Toc73114715)

[Arkkitehtuuri 2](#_Toc73114716)

[Askeleet 3](#_Toc73114717)

[Perussyntaksi suhtkoht sama kuin Razorissa. 4](#_Toc73114718)

[Datan välitys frontin komponenttien välillä 6](#_Toc73114719)

[JSInterop, eli javascriptin kutsuminen c# koodista. 8](#_Toc73114727)

[Css 9](#_Toc73114728)

# Blazor WASM demo, eli erillisen API:n päälle rakennettu frontti

Perustuu ihan siihen default rakenteeseen, joka blazorissa tulee kun aloittaa uuden blazor-projektin

Source: <https://github.com/RoniNiklas/BlazorDemo>

Client: <https://github.com/RoniNiklas/BlazorDemo/tree/master/BlazorDemo/Client>

# Mikä:

Blazor = C# pohjainen frontend framework.

Blazor WASM = C# SPA framework.

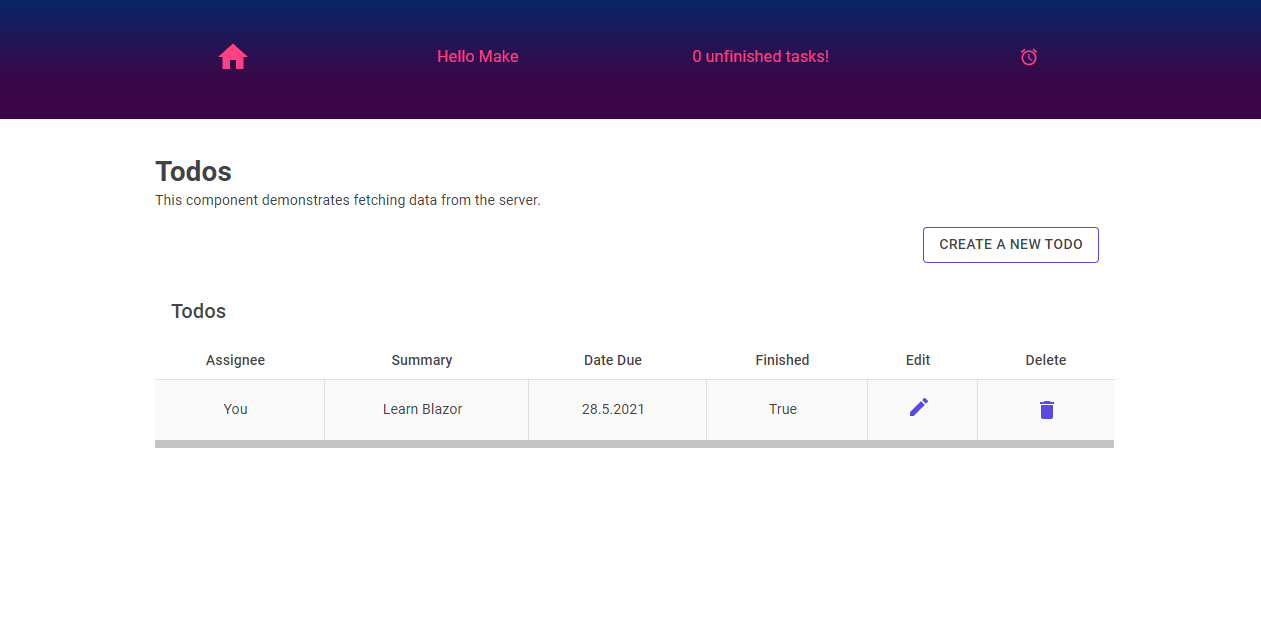
Suhtkoht samalla syntaksilla voi myös tehdä palvelinparsittuja blazor

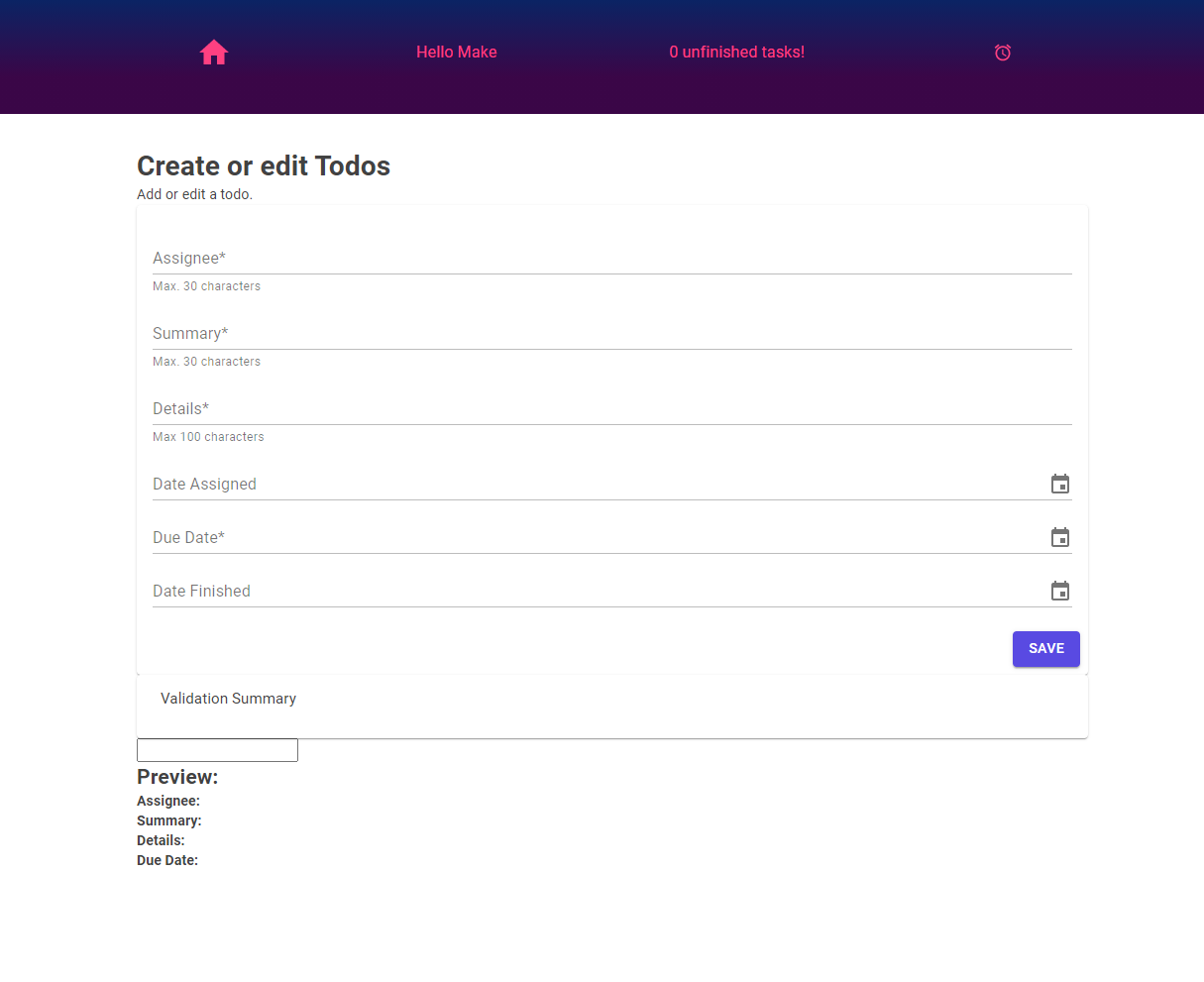
# Miksi:

* 1. C# syntaksi, C# typesafety
  2. C# koodin jako frontin ja backin välillä
  3. Ihan kiva syntaksi (melko sama kuin razorin)

# Kälileiskat:

Asettelu erillisen mudblazor-kirjaston komponenteilla. Asettelulla ei erityisesti tekemistä blazorin kanssa.





# Datan välitys palvelimen ja frontin välillä

## Arkkitehtuuri

Solutionissa kolme projektia

1. BlazorDemo.Client => frontti
2. BlazorDemo.Server => Backend
3. BlazorDemo.Shared => Jaetut luokat (viewmodelit), joita frontti ja back-käyttää

Etuna tässä on se, että frontti ja backi jakaa samat luokat BlazorDemo.Shared projektin kautta, joten datan välitys toimii luotettavasti niitten välillä, eikä täydy miettiä typescript-luokkien autogenerointia vue/react fronttiin jne..

Täten kenttien muuttuessa niin frontti ja backi pysyy kokoajan synkassa näitten jaettujen modeleitten suhteen. Koodia, esim. muuttujien validointia, ei myöskään tarvii duplikoida erikseen fronttiin/backiin.

Tässä käytössä erillinen käsintehty API, mutta oletettavasti mahdollista rakentaa esim. OData backend sillein, että shared-projektissa on ne odata-endpointtien käyttämät luokat.

## Askeleet

1. Luodaan jaettu viewmodeli tai vastaava jaettu luokka shared-projektiin

// Shared/TodoDetailsVM.cs

namespace BlazorDemo.Shared

{

public class TodoDetailsVM

{

public int? Id { get; set; }

[Required]

[StringLength(maximumLength: 100, ErrorMessage = "Details description is too long.")]

public string Details { get; set; } = "";

}

}

1. Backend controllerissa määritetään endpointin lähettämä luokka. Tässä tapauksessa TodoDetailsVM.

Server/Controllers/TodosController.cs

using BlazorDemo.Shared;

// irrelevanttia koodia

// relevantti endpointti

[HttpGet("{id}")]

public ActionResult<TodoDetailsVM?> GetDetailsVM(int id)

{

var entity = \_todoStore.GetById(id);

if (entity != null)

{

return new TodoDetailsVM()

{

Id = entity.Id,

Details = entity.Details

};

}

return NotFound();

}

1. Frontissa haetaan data, ja parsitaan suoraan haluttuun luokkaan HTTPClientin toimesta.

Client/Pages/TodosCreateOrEdit.razor

@using BlazorDemo.Shared

// irrelevanttia koodia

// relevantti datan haku

@code {

private TodoDetailsVM? Todo { get; set; }

protected override async Task OnInitializedAsync()

{

Todo = await Http.GetFromJsonAsync<TodoDetailsVM?>($"http://localhost:5000/todos/{Id}", cancellationToken: default);

}

}  
  
4. Voila. Frontti ja backi käyttävät samaa modelia molemmissa päissä datan välitystä ilman erityistä monimutkaisuutta.

Perussyntaksi suhtkoht sama kuin Razorissa.

Alla luontinäkymäsivun koodi Pages/TodoCreateOrEdit.razor

// Määritetään urli, josta näkymä löytyy, ja urlista poimittavat parametrit ja niiden tyyppi

@page "/todos/createOrEdit/{Id:int?}"

// @-merkki ennen koodin puolelta kutsuttuja muuttujia ja using/inject statementteja

@using BlazorDemo.Shared

@using MudBlazor

@using System.Globalization

@using BlazorDemo.Client.Components

@inject NavigationManager NavigationManager

@inject HttpClient Http

// Perus html:ää voi tunkea razor tiedostoon ihan normaalisti ilman kommervenkkejä

<h1>Create or edit Todos</h1>

<p>Add or edit a todo.</p>

// @ merkki, ja perus if-lauseita jos näkymä rakentuu ehdollisesti

@if (Loading)

{

<p><em>Loading...</em></p>

}

else if (Todo == null)

{

<p><em>Not found!</em></p>

} else {

// Custom-komponenttien kutsuminen parametreilla suhtkoht. samalla tavalla kuin Vue:ssa ja reactissa. Komponenttien namespace pitää importtaa, jos ne eivät ole samassa kuin page.

// Käytössä MudBlazor-komponenttikirjasto näkymää kaunistamaan, joka tietty ehkä oli väärä valinta demoon, koska sillä on aika paljon noita komponenttien parametreja, joita ei perus html:ssä ole

<**EditForm** **Model**="@Todo" **OnValidSubmit**="OnValidSubmit">

<**DataAnnotationsValidator** />

<**MudCard**>

<**MudCardContent**>

<**MudTextField** **Label**="Assignee" **HelperText**="Max. 30 characters" **Required**="true"

**@bind-Value**="Todo.Assignee" **For**="@(() => Todo.Assignee)" />

<**MudTextField** **Label**="Summary" **HelperText**="Max. 30 characters" **Required**="true"

**@bind-Value**="Todo.Summary" **For**="@(() => Todo.Summary)" />

<**MudTextField** **Label**="Details" **HelperText**="Max 100 characters" **Required**="true"

**@bind-Value**="Todo.Details" **For**="@(() => Todo.Details)" />

<**MudDatePicker** **Label**="Date Assigned" **Editable**="true" **Culture**="@FinnishCulture" **DateFormat**="dd/MM/yyyy"

**@bind-Date**="Todo.DateAssigned" />

<**MudDatePicker** **Label**="Due Date" **Editable**="true" **Culture**="@FinnishCulture" **DateFormat**="dd/MM/yyyy" **Required**="true"

**@bind-Date**="Todo.DueDate" />

<**MudDatePicker** **Label**="Date Finished" **Editable**="true" **Culture**="@FinnishCulture" **DateFormat**="dd/MM/yyyy"

**@bind-Date**="Todo.DateFinished" />

</**MudCardContent**>

<**MudCardActions**>

<**MudButton** **ButtonType**="ButtonType.Submit" **Variant**="Variant.Filled" **Color**="Color.Primary" **Class**="ml-auto">

Save

</**MudButton**>

</**MudCardActions**>

</**MudCard**>

<**MudExpansionPanels**>

<**MudExpansionPanel** **HideIcon**="true" **Text**="Validation Summary" **IsExpanded**="true">

<**MudText** **Color**="@Color.Error">

<**ValidationSummary** />

</**MudText**>

</**MudExpansionPanel**>

</**MudExpansionPanels**>

// Perus html-demo

// inputtien @bind-Value sitoo inputin muuttujaan, jolloin inputin arvon muuttaminen päivittää sitä muuttujaa, ja komponentti päivittyy sen mukana

<input **@bind-value**="Todo.Assignee" />

// Alakomponentti, joka ottaa parametrina Todo-muuttujan, päivittyy automaattisesti parametrin muuttuessa

<**TodoPreview** **Todo**="Todo" />

</**EditForm**>

}

// Isommat koodinpätkät @code {} sisään. Määritetään komponentin/näkymän parametrit ja sisäiset muuttujat ja metodit jne.

@code {

[Parameter]

public int? Id { get; set; }

private TodoDetailsVM? Todo { get; set; }

private bool Loading = true;

private CultureInfo FinnishCulture = CultureInfo.GetCultureInfo("fi");

protected override async Task OnInitializedAsync()

{

try

{

if (Id.HasValue)

{

// HTTPClient hakee ja parsii haetun datan halututtuun luokkaan

Todo = await Http.GetFromJsonAsync<TodoDetailsVM?>($"http://localhost:5000/todos/{Id}", cancellationToken: default);

} else {

Todo = new TodoDetailsVM();

}

Loading = false;

} catch (Exception e)

{

Console.WriteLine(e);

}

}

private async void OnValidSubmit(EditContext context)

{

// HTTPClient parsii ja lähettää luokan oikeassa muodossa

var result = await Http.PostAsJsonAsync("http://localhost:5000/todos/", Todo);

if (result.IsSuccessStatusCode) {

// NavigationManager hoitaa navigoinnin blazorrouterin kautta  
   
 NavigationManager.NavigateTo("http://localhost:5001/");

} else

{

Console.WriteLine("Server refused request");

}

}

}

# Datan välitys frontin komponenttien välillä

## Dataa voi passata parametrina vanhemmalta lapselle, jolloin lapsi päivittyy automaattisesti parametrin muuttuessa.

Parent Pages/TodoCreateOrEdit.razor: Kutsutaan komponenttia parametrilla

<**TodoPreview** **Todo**="Todo" />

Child Components/TodoPreview.razor: Napataan parametri @code-osiossa.

## <div>

## <strong>Assignee: </strong><span>@Todo.Assignee</span>

## </div>

@code {

[Parameter]

public TodoDetailsVM Todo { get; set; } = new TodoDetailsVM();

}

## Voi käyttää keskitettyä Storea, jota useampi komponentti käyttää samanaikaisesti. Tässä kolme askelta.

Käytetty store itse Stores/TodoStore.cs-tiedostossa => Perus c# luokka. Datan välitystä varten httpclient injektiona.

namespace BlazorDemo.Client.Stores

{

public class TodoStore

{

HttpClient Client { get; set; }

public List<TodoTableVM> TableVMs { get; set; } = new List<TodoTableVM>();

public int UnfinishedTasks

{

get

{

return TableVMs.Where(item => item.Finished == false).Count();

}

}

public TodoStore(HttpClient client)

{

Client = client;

}

// Määrittämätön Action OnChange muuttuja, joka automaagisesti toimii ilman, että sitä määritetään mitenkään. En ymmärrä, mutta toimii.

public event Action OnChange;

private void NotifyStateChanged() => OnChange?.Invoke();

public async Task FetchTableVMs()

{

TableVMs = await Client.GetFromJsonAsync<List<TodoTableVM>>( "http://localhost:5000/todos", cancellationToken: default) ?? new List<TodoTableVM>();

// Storen muuttujien muuttaminen itsessään ei päivitä storea käyttäviä komponentteja, vaan pitää kutsua notifyStateChanged metodia, joka sitten päivittää linkitetyt komponentit

NotifyStateChanged();

}  
 }  
}

Store pitää lisätä program.cs:ään, jotta se on saatavilla komponenteille injektiota varten

Program.cs

builder.Services.AddScoped<TodoStore>();

Store pitää injektoida sitä käyttäviin komponentteihin, joiden pitää implementtaa IDisposable-luokka, ja rekisteröidä itsensä seuraamaan storea

Shared/NavMenu.razor

// ennen näkymää

@inject TodoStore TodoStore

// näkymässä storea kutsutaan @-syntaksilla kuten kaikkea muutakin

<div>

<MudText Color="Color.Secondary"> @TodoStore.UnfinishedTasks unfinished tasks! </MudText>

</div>

// @code-osiossa pitää määrittää komponentti seuraamaan todoStorea, ja määrittää komponentti lopettamaan todoStoren seuraamisen, kun komponentti disposataan.   
@code{

protected override void OnInitialized()

{

TodoStore.OnChange += StateHasChanged;

}

public void Dispose()

{

TodoStore.OnChange -= StateHasChanged;

}

}

# JSInterop, eli javascriptin kutsuminen c# koodista.

Pitää eka määrittää jäsää

Client/wwwroot/alert.js  
function AlertWithText(alertText) {

window.alert(alertText);

}

function ReturnText(alertText) {

return "invoked returnText with param: " + alertText

}

Sitten se pitää laittaa saataville index.html:ssä

Client/wwwroot/index.html

<head>

<script src="js/alert.js"></script>

</head>

Tämän jälkeen se on kutsuttavissa komponenteista injektoimalla IJSRuntime komponenttiin, ja kutsumalla sen InvokeAsync metodeja javascript funktioiden nimillä. Funktioiden nimiä tai palautusarvoja ei tarkisteta mitenkään, joten typojen vaara on suuri. InvokeAsync palauttaa js-funktion palautusarvon, ja InvokeVoidAsync ei välitä js:n palautusarvosta.

Client/Shared/NavMenu.razor

public async void JavascriptTest ()

{

var result = await JS.InvokeAsync<string>("ReturnText", "This is just to test js interop");

Console.WriteLine(result);

await JS.InvokeVoidAsync("AlertWithText", "This is just to test js interop");

}

# Css

voi tehdä scoped-mallisesti kuten vue:ssa laittamalla .css-asetukset komponentinNimi.razor.css-tiedostoon. Tällöin vain tuon kyseisen komponentin elementit seuraavat css-tiedoston sääntöjä. Salettiin saa scss ja sass jne. mutten jaksanut testata.  
