ASP.NET CORE

Blazor

Deze cursus is eigendom van VDAB Competentiecentra © versie: 28/03/2024

INHOUD

1 Inleiding 7

1.1 Doelstelling 7

1.2 Vereiste voorkennis 7

1.3 Nodige software 7

2 Blazor apps 9

2.1 Wat is Blazor? 9

2.2 Blazor hosting models 9

2.3 Blazor render modes 10

2.4 Blazor app templates 10

2.5 Projectstructuur 15

3 Blazor components 19

3.1 Onderdelen van een component 19

3.2 Een component toevoegen 20

3.3 Errors opvangen met ErrorBoundary 23

3.4 Oefening 24

3.5 Route parameters 24

3.6 Oefening 26

3.7 Geneste componenten 27

3.7.1 Een copyright-component 27

3.7.2 Component parameters 28

3.7.3 Child content 29

3.7.4 Templated components 29

3.7.5 Oefening 36

3.7.6 Attribute splatting 36

3.8 Refereren naar een childcomponent 38

3.9 Code in een aparte class opnemen 40

3.9.1 Code in een partial class ... 40
3.9.2 Code in een baseclass ... 40
3.10 Oefening ... 41
3.11 Cascading values ... 41
3.11.1 Voorbeeld met component parameters ... 41
3.11.2 Cascading values i.p.v. component parameters ... 43
3.11.3 Meerdere cascading values op hetzelfde niveau ... 44
3.11.4 Meerdere cascading values op verschillende niveaus ... 47
3.11.5 Cascading value by type ... 47
3.11.6 Vaste cascading values ... 48
3.11.7 Cascading values overriden ... 49
3.12 Oefening ... 49
4 Data binding ... 51
4.1 Two-way binding ... 51
4.2 Bind-event ... 52
4.3 Ongeldige waarden ... 52
4.4 Data binding tussen parent en child component via eventcallback ... 53
4.5 Oefening ... 56
5 Event handling ... 57
5.1 DOM events afhandelen ... 57
5.2 Default actions tegenhouden ... 58
5.3 Stop eventpropagation ... 59
6 Component Lifecycles ... 61
7 Local Storage en Session Storage ... 65
7.1 Session Storage ... 65
7.2 Local Storage ... 67
7.3 Objecten wegschrijven ... 69
7.4 Protected browser storage ... 70

7.5 Oefening ... 72
8 CSS isolation ... 73
8.1 Een eenvoudig voorbeeld ... 73
8.2 Het ::deep pseudo-element ... 73
9 Layouts ... 75
9.1 Default layout ... 75
9.2 Afwijken van de default layout ... 75
9.3 Geneste layouts ... 77
10 Forms en validation ... 79
10.1 EditForm ... 79
10.2 Inputcomponenten ... 81
10.3 InputRadio ... 83
10.4 InputFile ... 84
10.5 Een eigen validation attribute ... 86
10.6 Comparevalidator ... 86
10.7 Validatie van complex types en collection types ... 87
10.8 EditContext ... 89
10.9 Oefening ... 93
11 Routing ... 95
11.1 De Router component ... 95
11.2 Routing naar een component uit een Razor class library ... 95
11.2.1 De class library gebruiken vanuit een Interactive Server Blazor app ... 96
11.2.2 De class library gebruiken vanuit een WebAssembly Blazor app ... 97
11.2.3 De class library gebruiken vanuit een Static Server Blazor app ... 97
11.2.4 De class library gebruiken vanuit een Interactive Auto Blazor app ... 98
11.3 NavLink ... 98
11.4 NavigationManager ... 100

12 JavaScript interoperability ... 103
12.1 JavaScript oproepen vanuit C# ... 103
12.2 Oefening ... 106
12.3 .NET methods oproepen vanuit JavaScript ... 106
12.3.1 Oproepen van static methods ... 107
12.3.2 Oproepen van een method van een object-instance ... 108
12.4 JavaScript isolation ... 111
13 Security ... 113
13.1 AuthorizeView component ... 113
13.2 [Authorize] attribuut ... 114
13.3 AuthorizeRouteView ... 114
13.4 Security in een Blazor WebAssembly app ... 114
13.4.1 Een beveiligde Blazor app ... 115
13.4.2 De security database ... 116
13.4.3 De beveiligde app uitproberen ... 116
13.5 Security in een Blazor Server app ... 117
14 Een Web API aanspreken vanuit Blazor ... 119
14.1 Opzetten van de applicatiestructuur ... 119
14.2 Models ... 119
14.3 Web API ... 120
14.3.1 Opbouw van de database Contacten ... 120
14.3.2 De Contacten-repository ... 121
14.3.3 De API-controller ... 123
14.4 Presentatie in Blazor WebAssembly ... 126
14.4.1 ContactenLijst ... 126
14.4.2 Een detailpagina ... 129
14.4.3 Contactgegevens wijzigen ... 131
14.4.4 Een contact verwijderen ... 134
14.4.5 Een contact toevoegen ... 136

14.5 Virtualization en lazy loading 137
14.5.1 Extra contacten 137
14.5.2 Virtualization 138
14.5.3 Lazy loading 140
15 Voorbeeldoplossingen oefeningen 145
15.1 Dobbelen 145
15.2 Dobbelen versie 2 145
15.3 Boekenlijst 146
15.4 Dobbelen versie 3 147
15.4.1 Met een partial class 147
15.4.2 Met een baseclass 148
15.5 Cascading value 148
15.6 Data binding 149
15.7 Protected Browser Storage 149
15.8 EditForm 150
15.9 JavaScript vanuit .NET oproepen 151

1 Inleiding

1.1 Doelstelling

In deze cursus leer je Single Page Applications maken met ASP.NET Core Blazor en .NET 8.

1.2 Vereiste voorkennis

Voor deze cursus is volgende voorkennis vereist :

- C#
- HTML5/CSS
- Javascript
- ASP.NET Core MVC
- EF Core

1.3 Nodige software

Bij de aanvang van de cursus veronderstellen we dat volgende software geïnstalleerd is :

- Visual Studio 2022
- SQL Server Management Studio

Deze cursus is gebaseerd op Visual Studio 2022 versie 17.9.4 en .NET 8.0.202. Je kan de versie van jouw Visual Studio raadplegen in Visual Studio via het menu Help – About Microsoft Visual Studio. De versie van jouw .NET SDK kan je opvragen in een commandvenster met het commando `dotnet --version`.

Verder werken we in deze cursus met het EntityFrameworkCore versie 8.0.2.

De kans bestaat dat er bij het doornemen van deze cursus reeds recentere versies van Visual Studio, .NET 8 en/of van EntityFrameworkCore bestaan. Misschien ziet een scherm er anders uit dan in de cursus weergegeven of misschien wordt er een recentere versie van een NuGet-package voorgesteld. Als je twijfelt welke versie je moet gebruiken, vraag dan raad aan jouw instructeur.

2 Blazor apps

In dit hoofdstuk krijg je een overzicht van wat Blazor is en hoe je een Blazor app maakt.

2.1 Wat is Blazor?

Blazor is een .NET webframework waarmee je een **Single Page Application** (SPA) kan maken. Om de user interface te maken van een Blazor applicatie gebruik je razorcode. Net zoals in een ASP.NET Core MVC webapplicatie dus. De applicatielogica schrijf je in C#.

Wat Blazor uniek maakt is dat je de taal **C#** ook gebruikt om client-side te programmeren. Dit in tegenstelling tot andere SPA frameworks zoals Angular en Knockout die hiervoor TypeScript of JavaScript gebruiken.

Blazor is ook anders dan bijvoorbeeld Silverlight, een oudere Microsoft technologie, waarbij je een browser plugin moest installeren om code te kunnen uitvoeren in de browser. Om een Blazor app te kunnen gebruiken in je browser heb je geen extra plugin nodig.

Om dit mogelijk te maken gebruikt Blazor **WebAssembly**. Dit is een instructieset en is net als HTML, CSS en JavaScript een web standaard, ondersteund door alle moderne browsers.

Een Blazor app kan ook op de server uitgevoerd worden. Page updates worden dan via een Signal-R verbinding naar de client gestuurd. We spreken dan van een **Blazor Server** app.

WebAssembly, de sleutel tot succes?

Aanvankelijk werd JavaScript-code voor uitvoering naar de browser gestuurd en daar geparset, gecompiled én ge-JIT-compiled. Dit zorgde voor heel wat vertraging bij uitvoering van de JavaScript-code. Door de JavaScript-code reeds op de webserver te parsen en compilen naar WASM (WebAssembly-code) werd heel wat tijd gewonnen.

.NET gebruikt Mono om bijvoorbeeld Xamarin-toepassingen te draaien op Windows, Android en iOS. Mono is geschreven in C++ en C++ kan ook WASM genereren. Via Mono kan .NET dus code omzetten naar WASM en laten uitvoeren in een browser.

2.2 Blazor hosting models

Er bestaan dus twee zogenaamde Blazor *hosting models* : Blazor WebAssembly en Blazor Server. We bekijken even de voor- en nadelen van beide hosting models.

Blazor WebAssembly

Een Blazor WebAssembly app draait volledig in de browser. Dus van zodra de app in de browser is geladen heeft het geen connectie meer nodig met de server. De app kan dus in principe zelfstandig verder draaien wanneer de webserver wordt uitgeschakeld. Hierdoor is er minder netwerkverkeer tussen browser en webserver en wordt de server ook minder belast. Voor het ophalen van (nieuwe) data is uiteraard wél nog internetconnectie nodig.

Er zijn ook nadelen. De opstart bijvoorbeeld is een stuk trager omdat alle code eerst moet gedownload worden naar de client. We lossen dit gedeeltelijk op met assembly lazy loading (zie hoofdstuk **Fout! Verwijzingsbron niet gevonden.**).

Blazor Server

Blazor Server apps hebben dan weer het voordeel dat ze iets sneller laden. Ook iets oudere browsers die niet overweg kunnen met Blazor WebAssembly kunnen Blazor Server apps wél draaien omdat er geen gebruik wordt gemaakt van de nieuwere technologie WebAssembly.

Blazor Server apps vergen dan weer meer inspanningen van de webserver op het vlak van geheugengebruik om alle sessiongegevens van de clients bij te houden.

Bij elke interactie van de gebruiker kijkt Blazor welk stukje van de pagina er moet gewijzigd worden. Om deze wijzigingen, zogenaamde Diff's, door te sturen naar de client gebruikt Blazor een SignalR-verbinding tussen client en server. Ook dit zorgt voor de nodige workload voor de server.

De Signal-R communicatie vertraagt ook de userinteractie met de applicatie. De zogenaamde UI latency of de tijd die nodig is om een scherm te updaten na een interventie van de gebruiker is dan ook een stuk hoger bij Blazor Server apps dan bij een Blazor WebAssembly app.

Opmerking

Er bestaat ook een variant op een Blazor WebAssembly app die ook een ASP.NET Core backend heeft op de server. We spreken dan van een ASP.NET Core hosted Blazor WebAssembly app.

2.3 Blazor render modes

Je bepaalt het Blazor hosting model van een applicatie via een Blazor render mode. Tot .NET 7 kon je kiezen tussen de render modes Blazor WebAssembly en Blazor Server. De render mode gold dan meteen voor de volledige applicatie.

Sinds .NET 8 kan je voor elk deel van jouw applicatie, zogenaamde Blazor componenten, afzonderlijk aangeven welke render mode er wordt gebruikt. De render mode Blazor Server wordt sinds .NET 8 ook wel **Interactive Server** genoemd, de render mode Blazor WebAssembly heet voortaan **Interactive WebAssembly**.

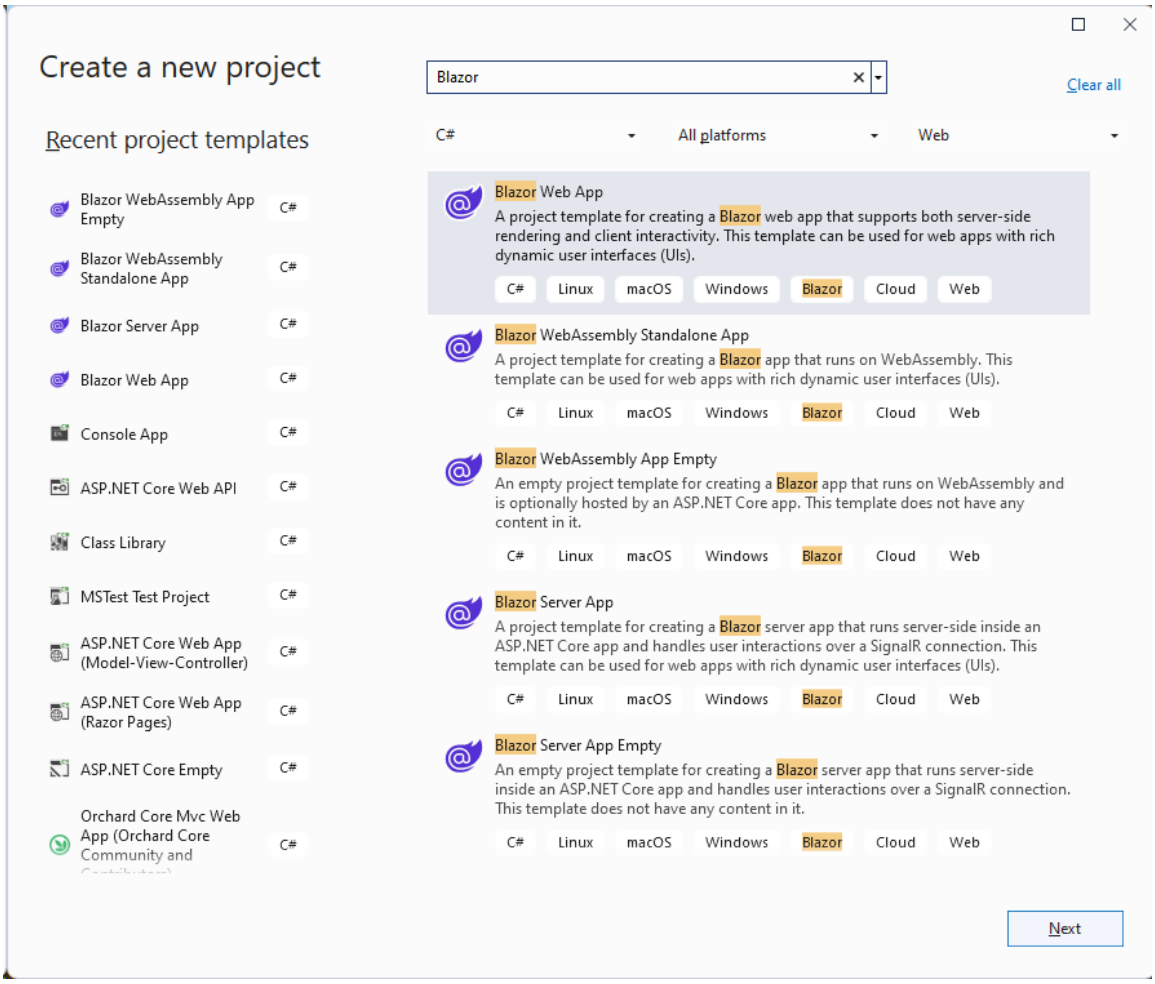
Er zijn in .NET 8 ook enkele render modes bijgekomen, namelijk **Static Server** en **Interactive Auto**. Static Server wordt gebruikt voor puur statische websites zonder dynamische content. Omdat het laden van een Blazor WebAssembly app even kan duren heeft men de render mode **Interactive Auto** geïntroduceerd. Hier wordt eerst de render mode Interactive Server gebruikt. Wanneer alles geladen is op de client wordt er automatisch overgeschakeld op Interactive Webassembly. Interactive Auto is dus een mengvorm van Interactive Server en Interactive WebAssembly.

2.4 Blazor app templates

Visual Studio biedt je enkele project templates aan waarmee je een Blazor app kan maken.

Wanneer je in Visual Studio 2022 een nieuw project aanmaakt en je tikt in het zoekvak bovenaan de term 'Blazor' dan krijg je 5 verschillende templates te zien.

- Start Visual Studio en maak een nieuw project aan.
- Tik in het zoekvak de term *Blazor*. Je krijgt onderstaand resultaat.



We overlopen even deze 5 templates.

Blazor Server App : met deze template maak je een Server Side Blazor app (interactive server) voor .NET 6 of .NET 7. De gekozen render mode geldt voor de volledig app.

Blazor Server App Empty : dit is eveneens een template voor een Server Side Blazor app voor .NET 6 of .NET 7 maar dan zonder voorbeeldpages.

Blazor WebAssembly Standalone App : met deze template maak je een Blazor WebAssembly app voor .NET 6, .NET 7 of .NET 8. De render mode geldt voor de volledige app.

Blazor WebAssembly App Empty : dit is eveneens een template voor een Blazor WebAssembly app maar enkel voor .NET 7 en zonder voorbeeldpages.

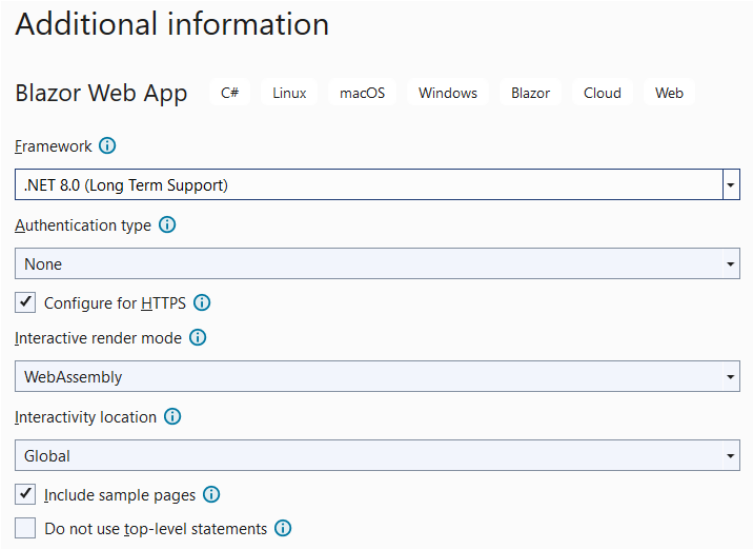
Blazor Web App : met deze template heb je de meeste vrijheid. Je kan kiezen welke interactive render mode je wil gebruiken en of die geldt voor de volledige app (global) of voor elke component individueel (per page/component). Het targetframework is .NET 8.

In deze cursus zullen we vooral de template Blazor Web App gebruiken. We maken nu enkele Blazor Apps met deze template.

- Kies in het venster *Create a new project* de template *Blazor Web App* en druk op *Next*.
- Geef het project de naam *BlazorWebAssembly.* Kies een geschikte locatie en noem de solution *Blazor.* Klik op *Next.*

In het volgende scherm kan je kiezen welk .NET framework je target, of er al dan niet security moet toegevoegd worden, kies je de interactive render mode en of die al dan niet geldt voor de volledige app.

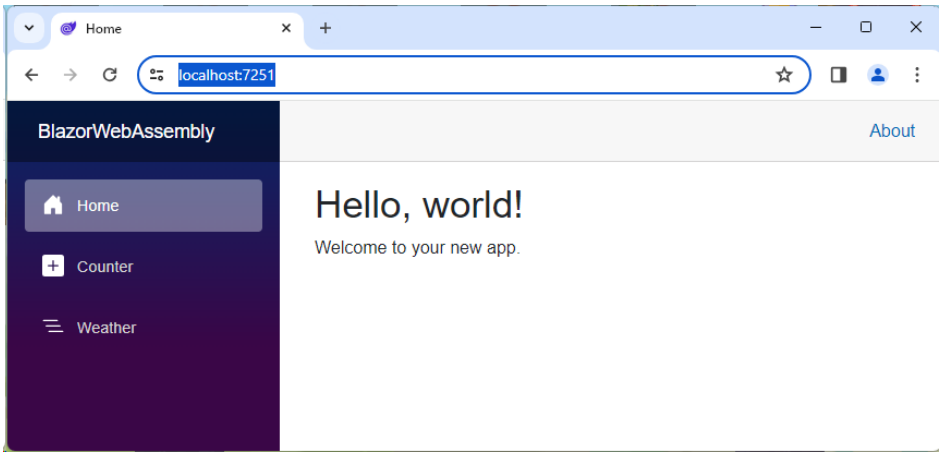
- Laat de default opties staan, behalve de Interactive render mode, die zet je op *WebAssembly.* Deze laten we gelden voor de volledige app (Global). Klik daarna op *Create.*



We kunnen deze app alvast eens uitproberen.

- Start de applicatie.

Achter de schermen wordt de volledige app naar de browser gestuurd en daar uitgevoerd. In de browser zie je het volgende verschijnen :



Aan de linkerkant zie je onder de titel van de app een menu met drie menuitems : Home, Counter en Weather. De optie Home is momenteel geselecteerd en toont een welkomstboodschap.

We proberen nu de app verder te gebruiken zonder dat deze door de webserver wordt gehost. Op die manier bewijzen we dat een Blazor WebAssembly app probleemloos verder draait wanneer de verbinding met de webserver wordt onderbroken.

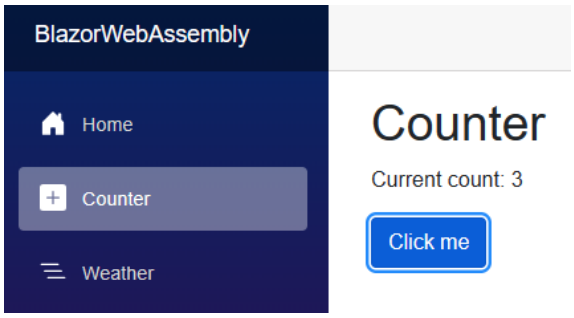
- Kopieer de URL in de browser.
- Open een andere browser.

- Plak de URL in de andere browser en probeer uit.
- Ga naar Visual Studio en stop de applicatie.
- Ga terug naar de andere browser en klik op het menuitem *Counter*.

De webapplicatie werkt probleemloos verder en toont het tweede app-onderdeel : Counter.

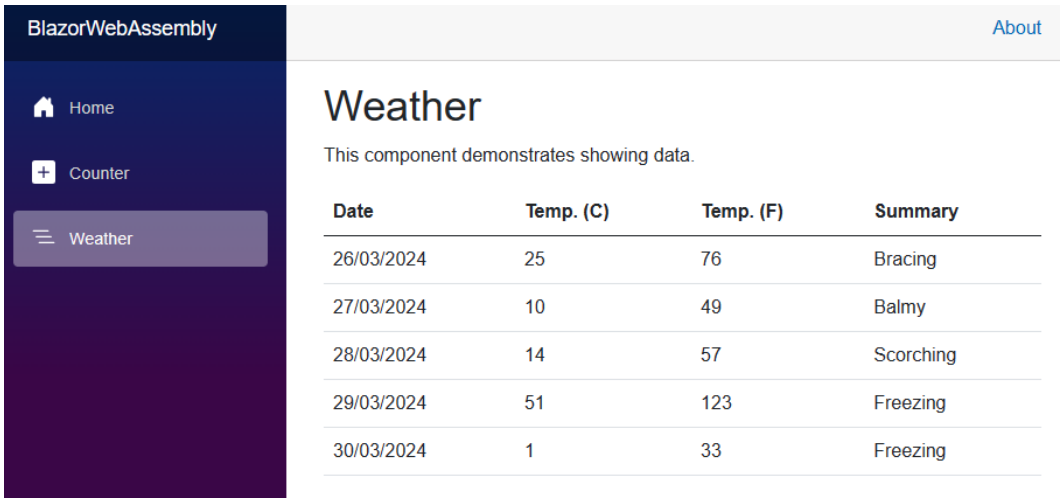
Deze mini-toepassing houdt een tellertje bij. Elke klik op de knop *Click me* verhoogt de teller. Misschien is het je niet opgevallen maar wanneer je de menuoptie Counter kiest gebeurt er geen refresh van de webpagina. Enkel het deel van de pagina dat moet worden bijgewerkt wijzigt.

- Klik een paar keer op de knop *Click me*. De waarde verhoogt, de pagina wordt niet volledig gerefresht maar wel bijgewerkt waar nodig (enkel het tellertje).



We gaan nu over naar het derde onderdeel van de app : *Weather*. Hier wordt static mockdata opgevraagd en getoond.

- Kies het menuitem *Weather.* Ook dit werkt probleemloos verder.



We proberen nu ook de Blazor Server app uit.

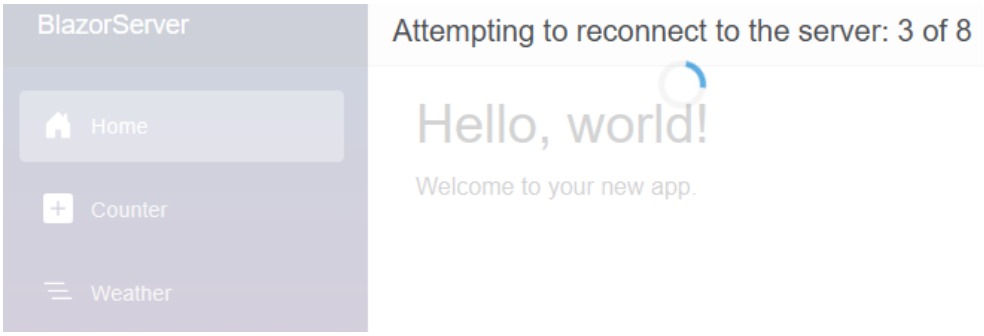
- Klik in de Solution Explorer met de rechtermuistoets op de naam van de solution en kies Add – New Project…
- Kies opnieuw een Blazor Web App en klik op Next.
- Je noemt het project BlazorServer. Klik op Next.
- Je stelt de Interactive render mode deze keer in op Server. De overige opties wijzig je niet. Klik op Create.
- Stel de Blazor Server app in als startup project.

- Start de applicatie.

De app ziet er voor de gebruiker exact hetzelfde uit als de WebAssembly app.

- Probeer de menuitems Counter en Weather uit.
- Kopieer de URL.
- Start een andere browser en plak er de URL in. Je krijgt eveneens de applicatie te zien.
- Stop de applicatie.

Je krijgt meteen onderstaande foutboodschap. De app werkt dus niet meer van zodra de hosting op de server stopt.



- Herstart de applicatie. Je krijgt opnieuw de applicatie te zien in de browser.

Een Blazor app met interactive render mode Interactive Auto lijkt sterk op een WebAssembly App. Het enige verschil is dat de app aanvankelijk als Blazor Server app wordt opgestart om een snellere start te garanderen.

We voegen een nieuwe Blazor app toe met interactive render mode Interactive Auto.

- Klik in de Solution Explorer met de rechtermuistoets op de naam van de solution en kies Add – New Project…
- Kies opnieuw een Blazor Web App en klik op Next.
- Je noemt het project BlazorAuto. Klik op Next.
- Je stelt de Interactive render mode in op Auto (Server and WebAssembly). De overige opties wijzig je niet. Klik op Create.
- Stel de app BlazorAuto in als startup project.
- Start de applicatie. De applicatie draait nu in Server modus.
- Kopieer de URL.
- Start een andere browser en plak er de URL in. Je krijgt eveneens de applicatie te zien.
- Stop de applicatie in Visual Studio.
- De applicatie stopt (ook in de andere browser).
- Herstart de applicatie. Je kan de applicatie herladen in de andere browser.
- Stop de applicatie. De applicatie draait nu in WebAssembly modus en werkt zonder server.

Tot slot voegen we ook een Blazor App toe met interactive render mode *Static Server.*

- Klik in de Solution Explorer met de rechtermuistoets op de naam van de solution en kies Add – New Project…

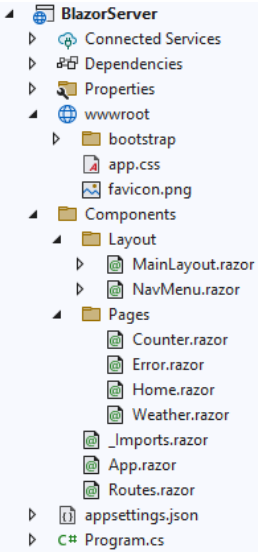
- Kies opnieuw een Blazor Web App en klik op Next.
- Je noemt het project BlazorStatic. Klik op Next.
- Je stelt de Interactive render mode in op None. De overige opties wijzig je niet. Klik op Create.
- Stel de app BlazorStatic in als startup project.
- Start de applicatie.

De applicatie BlazorStatic bevat als sample pages enkel een Homepage en een Weatherpage. Deze keer is er geen component Counter omdat interactiviteit met de user niet mogelijk is. De gebruiker kan dus niet op allerlei knoppen klikken of sliders aanpassen die de weergave van de pagina beïnvloeden. Wat er wél nog mogelijk is zijn HTML-forms. Die zorgen echter wel voor een refresh van de pagina, dit in tegenstelling tot de interactieve knoppen en controls in een Blazor component.

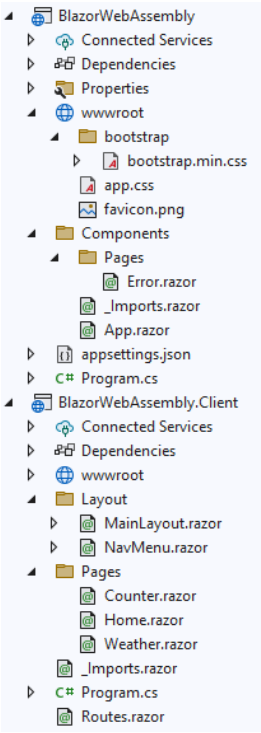
2.5 Projectstructuur

We hebben nu gezien hoe de sample pages er uit zien en hoe deze zich gedragen. We vergelijken nu de projectstructuur van de verschillende soorten Blazor apps.

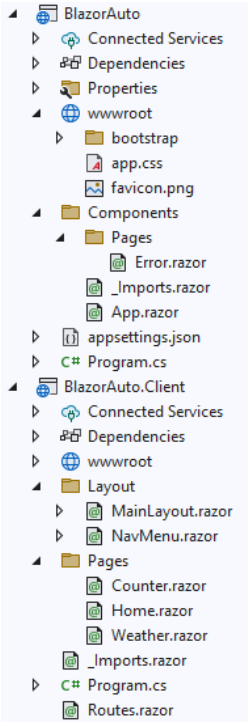
Blazor Server



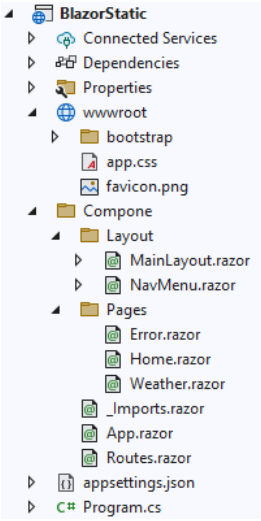
Blazor WebAssembly



Blazor Auto



Blazor Static



Het valt meteen op dat wanneer je kiest voor de interactive render modes *Interactive Server* en *Static Server* je slechts over één project beschikt. Bij de interactive render modes *Interactive WebAssembly* en *Interactive Auto* is de app verdeeld over twee projecten. Bij deze twee soorten Blazor apps moet het interactieve gedeelte van de app volledig op de client kunnen draaien. Daarom is dit in een apart project opgenomen waarvan de naam eindigt op *.Client*.

Verder zijn de Blazorprojecten heel gelijkaardig opgebouwd. In de folder *wwwroot* vinden we telkens dezelfde afbeeldingen en css-bestanden terug. De folder Components bevat Blazorcomponenten die

een pagina of een deel ervan voorstellen. Bij de render modes *WebAssembly* en *Interactive Auto* zijn deze componenten zoals gezegd verdeeld over twee projecten.

Tenslotte is er ook telkens een bestand *Program.cs*. In dit opstartbestand zie je dat telkens de service die instaat voor het verwerken van RazorPages wordt geladen. Bij een Blazor Server app komt daar ook nog de service *InteractiveServer* bij. In een Blazor WebAssembly app is dit de service *InteractiveWebAssembly.* In een Blazor app met interactive render mode InteractiveAuto worden beide services toegevoegd.

Beide apps beschikken dus over een aantal nagenoeg identieke razor-bestanden. Dit zijn Blazor componenten. Daarvan is **App.razor** de root component. Dit bestand ziet er in een Blazor Server app bijvoorbeeld zo uit :

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <base href="/" />
    <link rel="stylesheet" href="bootstrap/bootstrap.min.css" />
    <link rel="stylesheet" href="app.css" />
    <link rel="stylesheet" href="BlazorServer.styles.css" />
    <link rel="icon" type="image/png" href="favicon.png" />
    <HeadOutlet @rendermode="InteractiveServer" />
</head>

<body>
    <Routes @rendermode="InteractiveServer" />
    <script src="_framework/blazor.web.js"></script>
</body>

</html>
```

In de body van deze component wordt de component Routes.razor ingevoegd. De tag `<HeadOutlet>` is een placeholder voor content uit andere componenten die uiteindelijk in de headtag van de page moet opgenomen worden.

In de component **Routes.razor** zie je dat wanneer er een component wordt gevonden met een route die matcht met de ingetikte URL deze component wordt weergegeven met de layout MainLayout. Standaard krijgt in de gevonden component de tag h1 te focus.

De inhoud van Routes.razor :

```
<Router AppAssembly="typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="routeData" DefaultLayout="typeof(Layout.MainLayout)" />
        <FocusOnNavigate RouteData="routeData" Selector="h1" />
    </Found>
</Router>
```

De component **MainLayout.razor** is dus een layout-bestand dat aan de linkerkant van het scherm een navigatiebalk toont via de component *NavMenu.* Verder zie je ook een placeholder *Body.* Op die plaats wordt de inhoud van de gevraagde pagina in de layout ingevoegd.

In **NavMenu.razor** vinden we eerst razor-code voor een menu met twee of drie menuitems : Home, Counter en Weather. Het Counter-menuitem is niet aanwezig bij de rendermode Static Server. De menuitems brengen ons respectievelijk naar de homepagina, een Counter- en een Weather-component.

De componenten **Home.razor**, **Counter.razor** en **Weather.razor** heb je al aan het werk gezien. De component Counter is een interactieve component waarin je een tellertje kan verhogen. De component Weather toont na een kleine pauze, waarin een Loading-indicator wordt weergegeven, de weersgegevens van de komende 5 dagen. Deze gegevens worden random gegenereerd. De code hiervoor staat in een @code-block. Zoals je weet uit de cursus Razor Pages kan deze code ook in een apart bestand ondergebracht worden.

De component **_Imports.razor** bevat een aantal using-statements zodat we die statements niet in elke component apart moeten toevoegen.

3 Blazor components

Een Blazor app is opgebouwd uit Blazor components. In dit hoofdstuk zie je hoe een Blazor component is samengesteld en leer je zelf een blazor component maken. Je leert ook hoe je de routing naar een Blazor component instelt, eventuele parameters of attributen toevoegt en nog veel meer. We bekijken eerst de inhoud van de componenten Home, Counter en Weather, die standaard meegeleverd worden bij een nieuwe Blazor app. Daarna maken we zelf een Blazor component.

3.1 Onderdelen van een component

Een Blazor app start standaard met het tonen van een soort homepage. De inhoud ervan wordt bepaald door de component *Home.razor*. Deze component bevat volgende code :

```
@page "/"

<PageTitle>Home</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app.
```

Helemaal bovenaan staat een `@page` directive. Deze directive geeft aan dat deze component kan weergegeven worden door te browsen naar de url `"/"`. De string na `@page` geeft dus de route aan. Verderop in dit hoofdstuk leer je meer over Routing.

Opgelet : in Blazor moet je in de @page-directive een string opnemen die de route bepaalt. Hierin verschilt Blazor met Razor Pages. Daar kan deze string weggelaten worden en kan je gewoon `@page` noteren. De route wordt dan impliciet bepaald door de folder en de naam van de Razor Page. Hier kan dit dus niet.

De tag `<PageTitle>` bepaalt de titel in de browsertab. Verder krijgen we een h1-header en een welkomstboodschap te zien.

Kijken we naar een andere Blazor component, *Counter.razor,* dan zien we al wat meer code :

```
@page "/counter"
<PageTitle>Counter</PageTitle>
<h1>Counter</h1>
<p role="status">Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
@code {
    private int currentCount = 0;

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

De `@page` directive geeft aan dat deze Blazor component te bereiken is via de url `"/counter"`. Onderaan zien we een code-blok, aangegeven met de directive `@code`. Een Blazor component kan meerdere dergelijke code-blokken bevatten maar ook geen enkele (zoals in *Home.razor*). Wanneer er meerdere code-blokken zijn dan worden die bij compilatie gewoon samengevoegd.

Het code-blok van de Counter-component bevat een private variabele *currentCount* waarin we de waarde van een teller bewaren. Deze waarde kan in de component weergegeven worden met `@currentCount`. Daarnaast is er ook een method *IncrementCount()* die de waarde van deze teller met 1 verhoogt. In de view-code erboven wordt deze method met `@onclick="IncrementCount"` gekoppeld aan het click-event van de button.

Je kan de code uit het code-blok ook in een apart class-bestand opnemen. Je leert verderop in dit hoofdstuk hoe dit moet.

In de component *Weather,* beschreven in het bestand *Weather.razor*, zijn nog meer interessante zaken te zien. Helemaal onderaan het code-block wordt een WeatherForecast-class beschreven. Van deze class worden in de asynchrone method OnInitializedAsync() enkele instanties in de array *forecasts* gestopt. Van zodra deze array geladen is wordt deze getoond op het scherm. Is dit nog niet het geval dan wordt de tekst Loading… getoond. Je kan de Weather-component aanspreken via de route `/weather`.

3.2 Een component toevoegen

We voegen nu zelf een nieuwe component *Eenentwintig.razor* toe in de folder *Pages*. Deze Blazor component beschrijft een page waarop de gebruiker door het klikken op een button at random kaarten kan trekken. De getrokken kaarten worden naast elkaar getoond en de totale waarde van de kaarten wordt onder de kaarten weergegeven. We beperken ons hier tot het trekken van de kaarten en het tellen van de score. We hebben de scoreregels wel wat vereenvoudigd om ons te kunnen focussen op de essentie.

Als het niet specifiek staat vermeld dan zijn hieronder de voorbeelden in een Blazor Server app toegevoegd. Maar in principe is de code voor de Blazor apps met een andere rendermode net dezelfde. Kies je toch voor een andere rendermode, neem dan de component op in het Client-project waarvan de naam eindigt op .Client.

- Klik met de rechtermuistoets op de folder *Pages* en kies *Add – New Item…*
- Kies een *Razor Component*, noem deze *Eenentwintig.razor* en klik op *Add*.

> 👉 De naam van een Razor Component moet beginnen met een **hoofdletter**.
>
> 👉 Een component die een ganse pagina beslaat (zoals *Home.razor*, *Counter.razor* en *Weather.razor*) sla je best op in de folder ***Pages***. Je kan ook componenten maken voor kleinere, herbruikbare stukken layout. Die kan je best opslaan in een folder ***Components***. In Blazor apps die een afzonderlijk Client-project hebben zal je deze folder nog moeten maken.

- Wis alle code in *Eenentwintig.razor* en voeg onderstaande code toe :

```
@page "/eenentwintig"

<h1>Eenentwintig</h1>
<button class="btn btn-primary m-3" @onclick="TrekKaart">Trek een kaart</button>
<div class="mb-3">
    @foreach (var kaart in kaarten) {
        var pad = $"/images/{kaart}.png";
        <img src=@pad width="48" />
    }
</div>
```

```
<p class="alert alert-info">Jouw score : @currentScore</p>

@code {

    private int currentScore = 0;
    private List<string> kaarten = new List<string>();
    private static readonly string[] Soorten = { "H", "D", "C", "S" };
    private static readonly string[] Beelden = { "7", "8", "9", "10",
                                                 "J", "Q", "K", "A" };
    private static readonly byte[] Waarden = { 7, 8, 9, 10, 1, 2, 3, 11 };

    private string GetRandomCard()
    {
        Random random = new Random();
        var soort = Soorten[random.Next(0, 4)];
        var kaart = random.Next(0, 8);
        var beeld = Beelden[kaart];
        var waarde = Waarden[kaart];
        currentScore += waarde;
        return $"{beeld}{soort}";
    }

    private void TrekKaart()
    {
        kaarten.Add(GetRandomCard());
    }

}
```

De code start met een `@page` directive. Hiermee geven we aan dat deze component kan weergegeven worden door te browsen naar de url *…/eenentwintig*.

Om de score te kunnen bijhouden definiëren we in het `@code` blok een int-variabele *currentScore* die we op 0 initialiseren. De score wordt onderaan de pagina weergegeven met `@currentScore`.

De getrokken kaarten worden bewaard in een List<string> met de naam *kaarten*. De mogelijke soorten en afbeeldingen van de kaarten en hun bijhorende waarde worden gedefinieerd in static array's.

In de method *GetRandomCard()* wordt at random een soort en een afbeelding gekozen. De score wordt verhoogd met de waarde van de kaart. Om het niet te complex te maken krijgt een aas altijd de waarde 11. Tenslotte wordt de kaart in stringformaat teruggegeven. Deze string is van de vorm 7H (harten-zeven), AS (schoppenaas), …

Met de code `@onclick="TrekKaart"` koppelen we de method *Trekkaart()* aan de button. In die method roepen we de method *GetRandomCard()* op. Het resultaat, een nieuwe kaart, voegen we toe aan de kaarten-array.

Onder de button geven we alle getrokken kaarten weer. Een verwijzing naar static content zoals een image begint met een slash (/). Een tilde-slash notatie (~/) is niet toegestaan in Blazor.

Voor elke kaart voorzien we een image in een nieuwe folder *images.*

- Voeg in *wwwroot* een folder *images* toe.
- Kopieer in deze folder de afbeeldingen van de kaarten (zie oefenbestanden).

Tot slot voegen we voor deze nieuwe component een menuitem toe in *NavMenu.razor* en passen we het gebruik van de icons aan.

De icons die standaard in het navigatiemenu van een Blazor app worden gebruikt staan gedefinieerd in het bestand *Navmenu.razor.css*. Om vlot nieuwe menuitems met bijhorende icons te kunnen toevoegen zullen we gebruik maken van Bootstrap icons. Om die icons te kunnen gebruiken dienen we een link toe te voegen in *App.razor*.

- Voeg in *App.razor* in de headtag een extra link toe :

```
…
<head>
    …
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap-
icons@1.11.3/font/bootstrap-icons.min.css">
    …
</head>
…
```

In *NavMenu.razor* vervangen we de gebruikte icons door Bootstrap-icons en we voegen een extra -tag toe voor het nieuwe menuitem.

- Wijzig de classnamen in de span-tags binnen de NavLinks en voeg een extra menuitem toe.

```
…
<nav class="flex-column">
    <div class="nav-item px-3">
        <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
            <span class="bi bi-house-door-fill" aria-hidden="true">
            </span> Home
        </NavLink>
    </div>

    <div class="nav-item px-3">
        <NavLink class="nav-link" href="counter">
            <span class="bi bi-plus-square-fill" aria-hidden="true">
            </span> Counter
        </NavLink>
    </div>

    <div class="nav-item px-3">
        <NavLink class="nav-link" href="weather">
            <span class="bi bi-list-nested" aria-hidden="true"></span> Weather
        </NavLink>
    </div>

    <div class="nav-item px-3">
        <NavLink class="nav-link" href="eenentwintig">
            <span class="bi bi-heart" aria-hidden="true"></span> 21
        </NavLink>
    </div>
</nav>
</div>
```

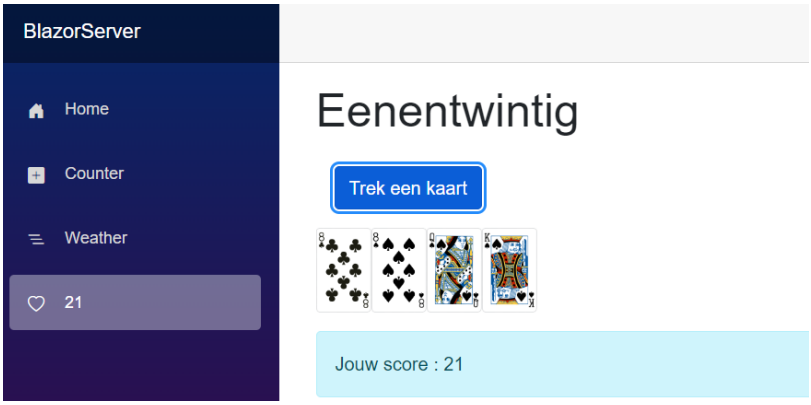
Om de iconen uit te lijnen met de menubeschrijvingen moeten we de css in NavMenu.razor.css een beetje aanpassen.

- Wijzig in NavMenu.razor.css de css-rule *top* in de css-class *bi* als volgt :

```
.bi {
    …
    top: -0.75rem;
}
```

In het hoofdstuk over Routing hebben we het nog uitgebreid over de NavLink-component.

Je kan de nieuwe component uitproberen :



3.3 Errors opvangen met ErrorBoundary

Wanneer zich een fout voordoet in een component dan kunnen we dit op een elegante manier opvangen door de component te omsluiten met een ErrorBoundary-component.

We proberen dit uit door in de component Eenentwintig een exception te throwen wanneer we meer dan 21 krijgen.

- Pas in Eenentwintig.razor de code in de method TrekKaart() als volgt aan :

```
private void TrekKaart()
{
    kaarten.Add(GetRandomCard());
    if (currentScore > 21)
    {
        throw new InvalidOperationException("Je bent verbrand!");
    }
}
```

We voegen de component *Eenentwintig* nu toe als subcomponent van de component *Home.razor* en bekijken het resultaat van onze sabotage.

- Wijzig de code in Home.razor als volgt : je voegt helemaal onderaan een component Eenentwintig toe.

```
…
Welcome to your new app.

<Eenentwintig />
```

- Start de app zonder debugging (Ctrl+F5) en trek kaarten totdat je boven de 21 uitkomt.

Je krijgt onderaan het browserscherm een foutboodschap *An unhandled error has occurred…*

An unhandled error has occurred. Reload

We vangen de fout nu op door de component Weather te omsluiten met een ErrorBoundary-component.

- Wijzig de code in Home.razor als volgt :

```
…
Welcome to your new app.

<ErrorBoundary>
    <ChildContent>
        <Eenentwintig />
    </ChildContent>
    <ErrorContent>
        <p class="alert alert-danger">Je bent verbrand</p>
    </ErrorContent>
</ErrorBoundary>
```

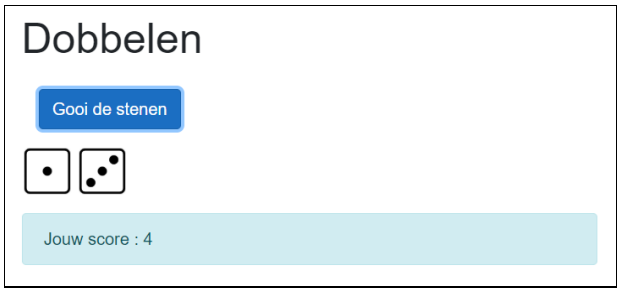
Binnen een ErrorBoundary-component staat een ChildContent en een ErrorContent. De inhoud van de ErrorContent wordt getoond van zodra er zich een fout voordoet binnen de inhoud van de ChildContent component.

- Probeer opnieuw uit zonder debugging. Je ziet bij een score boven de 21 een elegantere boodschap.

3.4 Oefening

Maak een nieuwe component *Dobbelen.* Na een klik op een knop worden twee dobbelstenen gegooid. Toon op het scherm de dobbelstenen en hun gezamenlijke waarde.

Een mogelijk resultaat :



3.5 Route parameters

Je kan aan de route van jouw component een parameter toevoegen. Als voorbeeld passen we de route van de Weather-component aan zodat de url */weather/2* de voorspelling weergeeft van overmorgen. De parameter geeft dus het aantal dagen in de toekomst weer.

Als je de waarde uit de route wil doorgeven aan een component moet je twee zaken doen :

1. In de page-directive een **parameter** toevoegen tussen accolades.
2. In het code-blok een **property** toevoegen voorzien van een [Parameter]-annotation.

We beginnen met de page-directive.

- Voeg in *Weather.razor* een extra page-directive toe :

```
@page "/weather"
@page "/weather/{dag}"
…
```

De component Weather kan nu op twee manieren opgeroepen worden : zonder en mét extra parameter. Dit maakt de parameter dus optioneel. Wil je de parameter verplicht maken dan verwijder je de route zonder parameter. Dit hoef je nu niet te doen, we behouden beide routes.

- In het code-blok voeg je nu onderstaande property toe :

```
[Parameter]
public string? Dag { get; set; }
```

Wanneer je nu browset naar /weather/2 dan zal de property *Dag* automatisch de waarde “2” krijgen.

We passen nu de layoutcode van de component aan.

- Voeg onderstaande vetgedrukte code toe :

```
…
@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    if (Dag != null)
    {
        var dag = int.Parse(Dag);
        var forecast = forecasts[dag-1];
        <h3>Forecast for @forecast.Date.ToShortDateString() (today + @dag day(s))
        </h3>
        <p>Temp. (C) : @forecast.TemperatureC</p>
        <p>Temp. (F) : @forecast.TemperatureF</p>
        <p>Summary : @forecast.Summary</p>
    }
    else
    {
        <table class="table">
        …
        </table>
    }
    …
```

We controleren eerst of de property *Dag* ingevuld werd door de routeparameter. Als dat het geval is converteren we de parameterwaarde naar een int. De index voor de gevraagde voorspelling is de waarde van *dag* - 1. Dus als de parameter “2” is dan willen we de waarde voor overmorgen en die staat in de array op positie 1. Daarna tonen we de voorspellingswaarden.

Is de routeparameter niet ingevuld – Dag is dan null – dan tonen we de ganse tabel.

- Probeer uit : start de app en kies in het menu *Weather.* Je ziet een 5-dagen voorspelling.
- Wijzig de url in de browser naar …/weather/2. Je ziet de voorspelling voor overmorgen.

Vullen we echter een niet-numerische waarde in voor de routeparameter *Dag* of een numerische waarde kleiner dan 1 of groter dan 5 dan krijgen we ofwel een FormatException ofwel een IndexOutOfRangeException.

Blazor beschikt over een beperkt aantal routeconstraints waarmee we alvast het eerste probleem kunnen oplossen.

- Wijzig de page-directive met de routeparameter als volgt :

```
@page "/weather/{dag:int}"
```

De constraint `:int` geeft aan dat de routeparameter *dag* een int-type moet zijn. Is dit niet het geval dan krijgen we de standaardboodschap voor een ongeldige URL : *Sorry, there's nothing at this address.*

Naast de int-constraint zijn er ook constraints voor de types bool, datetime, decimal, double, float, guid en long.

Nu we zeker zijn dat de routeparameter een int is kunnen we de code wat vereenvoudigen. We testen ook of de parameter een geldige waarde heeft : minimum 1 en maximum 5.

- Wijzig de code als volgt :

```
    …
    @if (forecasts == null) {
        <p><em>Loading...</em></p>
    }
    else
    {
        if (Dag > 0 && Dag < 6)
        {
            var forecast = forecasts[Dag - 1];
            <h3>Forecast for @forecast.Date.ToShortDateString() (today + @Dag day(s))
            </h3>
            <p>Temp. (C) : @forecast.TemperatureC</p>
            <p>Temp. (F) : @forecast.TemperatureF</p>
            <p>Summary : @forecast.Summary</p>
        }
        …
    }
}

@code {
    private WeatherForecast[] forecasts;

    [Parameter]
    public int Dag { get; set; }
    …
}
```

We wijzigen het type van de parameter-property *Dag* dus in een int. Omdat *Dag* een int-type heeft krijgt *Dag* automatisch de beginwaarde 0. In de layoutcode testen we of de waarde van deze property tussen 1 en 5 ligt. Als dat zo is dan tonen we de voorspelling van de gevraagde dag, is dit niet zo dan tonen we alle voorspellingen.

- Je kan de nieuwe code uitproberen.

3.6 Oefening

Pas de component *Dobbelen* als volgt aan. Je kan de component ook aanroepen met een int-parameter *aantal.* Deze parameter geeft aan met hoeveel dobbelstenen je speelt. Zonder parameter wordt er met 2 dobbelstenen gespeeld. Je hoeft niet te testen of het aantal negatief is of overdreven groot.

Resultaat van de url *…/dobbelen/3*

Tip : wanneer er geen routeparameter wordt meegegeven in de url dan initialiseer je het aantal dobbelstenen in een method *OnInitialized()* :

```
protected override void OnInitialized()
{
    base.OnInitialized();
    if (Aantal == 0) Aantal = 2;
}
```

3.7 Geneste componenten

De meeste componenten die we tot nu toe hebben bekeken beschrijven de layout van een ganse pagina. Je kan ook componenten maken die slechts een stukje van een pagina beschrijven. Dit soort componenten kan vervolgens in andere componenten (her-)gebruikt worden.

3.7.1 Een copyright-component

In deze paragraaf maken we een component voor een footer waarin we een copyright-tekst zullen weergeven. Deze footer kan dan op om het even welke component worden opgenomen.

Aangezien deze component geen volledige page beslaat zullen we de component in een folder *Components* plaatsen. Afhankelijk van de gekozen rendermode beschikt jouw Blazor app reeds over een dergelijke folder. In Blazor apps die een aparte .Client-project hebben zal je deze folder zelf moeten toevoegen in het .Client-project.

- Voeg indien nodig een nieuwe folder *Components* toe.
- Klik met de rechtermuistoets op deze folder en kies *Add-New Item…*
- Kies een Razor Component en noem deze *CopyrightFooter.razor*.
- Wijzig de code in *CopyrightFooter.razor* als volgt (tik gerust jouw voornaam na het copyright-teken) :

```
<div class="alert alert-info">
    <div>&copy; Steven</div>
</div>

@code {

}
```

We voegen deze nieuwe component nu helemaal onderaan toe in de component *Eenentwintig.razor*.

- Net onder de weergave van de score voeg je de tag `<CopyrightFooter />` toe :

```
…
<p class="alert alert-info">Jouw score : @currentScore</p>
```

```
<CopyrightFooter />

@code {
…
```

Normaal wordt de nieuwe tag met een groen golvend streepje onderlijnd. Wanneer je de beschrijving van de warning leest zie je dat deze tag nog niet gekend zou zijn omdat de component tot een ongekende namespace behoort en er een using-statement moet toegevoegd worden. Dit klopt echter niet. In *_Imports.razor* staat reeds een dergelijk using-statement.

- Start de app en browse nu naar */eenentwintig*. De copyright-component wordt onder de score weergegeven.

3.7.2 Component parameters

In een vorige paragraaf hebben we de component *Weather* voorzien van een parameter om aan te geven van welke dag in de toekomst we de voorspelling willen zien. De waarde voor de parameter *Dag* werd doorgegeven via de route.

Je kan echter ook de component parameter een waarde geven via een component attribuut. Je voegt in een bestaande component de weersvoorspelling voor morgen toe met volgende code :

```
<Weather Dag="1"/>
```

Bij wijze van oefening voegen we nu ook een parameter toe aan de copyrightfooter component. De naam van de auteur in deze component is nu statisch. We kunnen de naam variabel maken door de naam door te geven aan de component als een parameterwaarde. Om dit mogelijk te maken moeten we een property toevoegen in de component, voorzien van een [Parameter]-annotation. In de layout-code vervangen we de naam door de waarde van de property.

- Wijzig de code in *CopyrightFooter.razor* als volgt :

```
<div class="alert alert-info">
    <div>&copy; @Auteur</div>
</div>

@code {

    [Parameter]
    public string? Auteur { get; set; }

}
```

We kunnen in *Eenentwintig.razor* de component nu voorzien van een attribuut Auteur en een bijhorende waarde.

- Wijzig de code in *Eenentwintig.razor* als volgt :

```
<CopyrightFooter Auteur="Steven"/>
```

- Probeer opnieuw uit.

3.7.3 Child content

De component *CopyrightFooter* is nu een self-closing tag en heeft enkel een attribuut Auteur. Het heeft verder geen content. Daar brengen we verandering in.

Om de component te kunnen voorzien van content voeg je een property *ChildContent* toe die je eveneens voorziet van een [Parameter]-annotation. Deze property is van het type ***RenderFragment***. Vervolgens kan je de content met de code `@ChildContent` een plaats geven in de layout.

- Wijzig de code in *CopyrightFooter.razor* als volgt :

```
<div class="alert alert-info">
    <div>&copy; @Auteur</div>
    <div>@ChildContent</div>
</div>

@code {

    [Parameter]
    public string? Auteur { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }
}
```

We voegen de *CopyrightFooter* nu toe in *Eenentwintig.razor* mét de childcontent die in de component moet weergeven worden. Deze content kan naast gewone tekst ook html-tags bevatten.

- Wijzig de code in *Eenentwintig.razor* als volgt :

```
<CopyrightFooter Auteur="Steven">
    <h4>Blazor rocks</h4>
</CopyrightFooter>
```

- Probeer uit.

De content die je doorgeeft aan de component kan je ook expliciet binnen een *ChildContent*-tag plaatsen. De code ziet er dan als volgt uit :

```
<CopyrightFooter Auteur="Steven">
    <ChildContent>
        <h4>Blazor rocks</h4>
    </ChildContent>
</CopyrightFooter>
```

3.7.4 Templated components

We kunnen aan een component ook één of meerdere html-fragmenten als parameter doorgeven door ze bij de oproep te noteren als **subelementen**.

Voorbeeld:

```
<BlogpostTemplate>
    <Kop>Voorbeeld templated component</Kop>
    <Inhoud>Blazor rocks !</Inhoud>
    <Footer>VDAB</Footer>
</BlogpostTemplate>
```

De component BlogpostTemplate is een component met de subelementen Kop, Inhoud en Footer.

Op deze manier wordt de component een soort template met op bepaalde plaatsen de variabele content uit de subelementen. Technisch gezien zijn de subtags Kop, Inhoud en Footer Render-Fragmenten die als parameter worden doorgegeven. De component noemen we een templated component.

Als eerste voorbeeld maken we bovenstaande templated component voor een blogpost. Bij de oproep van de templated component wordt de content van de titel, het artikel en de footer meegegeven.

- Maak in de folder *Pages* een nieuwe razor component en noem die *Blog.razor.*
- Voeg aan *NavMenu.razor* een extra menuitem *Blog* toe :

```
…
<div class="nav-item px-3">
    <NavLink class="nav-link" href="blog">
        <span class="bi bi-chat" aria-hidden="true"></span> Blog
    </NavLink>
</div>
…
```

- We maken nu de templated component. Maak in de folder *Components* een razor component *BlogpostTemplate*. Dit wordt het model voor een blogpost.
- Vervang de code in *BlogpostTemplate.razor* door het onderstaande :

```
<h2 style="font-style:italic;">@Kop</h2>
<hr />
<div class="blogpost">
    @Inhoud
</div>
<hr />
<p class="blogpostFooter">@Footer</p>

@code {

    [Parameter]
    public RenderFragment? Kop { get; set; }

    [Parameter]
    public RenderFragment? Inhoud { get; set; }

    [Parameter]
    public RenderFragment? Footer { get; set; }
}
```

De BlogpostTemplate-component heeft 3 Parameter-property's van het type RenderFragment : Kop, Inhoud en Footer. In de layout krijgen deze RenderFragmenten hun plaats tussen eventuele static content.

Deze BlogpostTemplate kan nu in een andere component opgenomen worden. In die component wordt de inhoud voor de RenderFragments bepaald door de content van de gelijknamige tags.

Voorbeeld :

```
<BlogpostTemplate>
    <Kop>Dit is de tekst voor de kop</Kop>
    <Inhoud>Hier komt de inhoud voor het RenderFragment inhoud</Inhoud>
    <Footer>De footertekst</Footer>
</BlogpostTemplate>
```

We voegen nu wat css toe om de blogpost op te maken. In de folder *wwwroot* vind je een bestand *app.css.* Hier kan je extra css aan toevoegen.

- Voeg in *app.css* onderstaande code toe voor extra opmaak.

```
.blogpost {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    font-style:italic;
    background-color:#e7e7e7;
    padding: 25px;
    border-radius: 10px;
}

.blogpostFooter {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    margin-bottom: 30px;
}
```

De BlogpostTemplate is klaar. We kunnen deze nu gebruiken in *Blog.razor.*

- Vervang de code in de component *Blog.razor* door onderstaande code :

```
@page "/blog"
<h1 style="text-align:center">- .NET Web Development Blog -</h1>
<hr />
<BlogpostTemplate>
    <Kop>&laquo;Blazor rules the world&raquo;</Kop>
    <Inhoud>Blazor is a Single Page Application development framework. The name Blazor
is a combination/mutation of the words Browser and Razor (the .NET HTML view
generating engine). The implication being that instead of having to execute Razor
views on the server in order to present HTML to the browser, Blazor is capable of
executing these views on the client.</Inhoud>
    <Footer>Blazor University - &copy; 2020</Footer>
</BlogpostTemplate>
<BlogpostTemplate>
    <Kop>ASP.NET Core MVC</Kop>
    <Inhoud>
        The ASP.NET Core MVC framework is a lightweight, open source, highly testable
presentation framework optimized for use with ASP.NET Core.<br />ASP.NET Core MVC
provides a patterns-based way to build dynamic websites that enables a clean
separation of concerns. It gives you full control over markup, supports TDD-friendly
development and uses the latest web standards.
    </Inhoud>
    <Footer>Microsoft - &copy; 2018</Footer>
</BlogpostTemplate>
<BlogpostTemplate>
    <Kop>Razor Pages</Kop>
    <Inhoud>ASP.NET Core Razor Pages is a page-focused framework for building dynamic,
data-driven web sites with clean separation of concerns. Based on the latest version
of ASP.NET from Microsoft - ASP.NET Core, Razor Pages supports cross platform
development and can be deployed to Windows, Unix and Mac operating systems.</Inhoud>
    <Footer>LearnRazorPages - &copy; 2020</Footer>
</BlogpostTemplate>

@code {

}
```

- Probeer uit : start de app en kies het menuitem *Blog*.

Het resultaat :

In een tweede voorbeeld maken we een templated component voor een table. Deze component kan dan gebruikt worden voor een artikeltabel, een personeelstabel,…

De component krijgt net als in het vorige voorbeeld via subelementen een aantal RenderFragments mee : één voor de tableheader (`<TableHeader>`), één voor de tablefooter (`<TableFooter>`) en één die een rij in de tabel beschrijft (`<RowTemplate>`). Dit laatste renderfragment is een speciaal render-fragment want het krijgt de gegevens van één object mee.

Daarnaast geven we aan de template ook een lijst met objecten mee (`Items`). Daarvoor voegen we een parameter property toe aan de template.

Voorbeeld van het gebruik van de TableTemplate :

```
<TableTemplate Items="artikels">
    <TableHeader>html voor de header van het table-element</TableHeader>
    <RowTemplate>html voor één artikel</RowTemplate>
    <TableFooter>html voor de footer-content</TableFooter>
</TableTemplate>
```

- Maak in de folder *Pages* een nieuwe razor component en noem die *Tables.razor.*
- Voeg aan *NavMenu.razor* een extra menuitem *Tables* toe :

```
…
<div class="nav-item px-3">
    <NavLink class="nav-link" href="tables">
        <span class="bi bi-list" aria-hidden="true"></span> Tables
    </NavLink>
</div>
…
```

- Maak in de folder *Components* een razor component *TableTemplate*.
- Vervang de aanwezige code door de onderstaande code :

```
@typeparam TItem

<table class="table table-responsive mb-4">
    <thead>
        <tr>@TableHeader</tr>
    </thead>
    <tbody>
        @foreach (var item in Items)
        {
            <tr>@RowTemplate(item)</tr>
        }
    </tbody>
```

```
    <tfoot>
        <tr>@TableFooter</tr>
    </tfoot>
</table>

@code {
    [Parameter]
    public RenderFragment? TableHeader { get; set; }

    [Parameter]
    public RenderFragment<TItem>? RowTemplate { get; set; }

    [Parameter]
    public RenderFragment? TableFooter { get; set; }

    [Parameter]
    public IEnumerable<TItem>? Items { get; set; }
}
```

Op de eerste lijn geven we met de `@typeparam` directive aan dat deze component een 'generic' templated component is. Dus een templated component dat een welbepaald type objecten zal weergeven. TItem is hier een soort alias voor het échte type van de objecten die via de parameter *Items* zullen doorgegeven worden.

In de layout van *TableTemplate* beschrijven we hoe de table er zal uitzien. In de hoofding van de tabel gebruiken we het RenderFragment *TableHeader*, in de footer het RenderFragment *TableFooter*. In de body van de tabel overlopen we alle elementen van de parameter *Items*. Het type van deze items kennen we nog niet bij de oproep van deze component dus gebruiken we de alias *TItem* om de parameter te beschrijven als een IEnumerable<TItem>. Binnen de foreach-lus gebruiken we een generic RenderFragment<TItem> *RowTemplate* dat één item als parameter meekrijgt om weer te geven.

Onze universele tabletemplate is nu klaar voor gebruik in de component *Tables.razor.* Daarin gaan we artikels uitlijsten.

- Maak in de root van het project een nieuwe folder *Models* toe.
- Voeg in de folder *Models* onderstaande class *Artikel* toe :

```
public class Artikel
{
    public int Id { get; set; }
    public string Naam { get; set; }
    public string Omschrijving { get; set; }
    public decimal Prijs { get; set; }
}
```

- Vervang de code in *Tables.razor* door het onderstaande :

```
@page "/tables"

<TableTemplate Items="artikels">
    <TableHeader>
        <th>Id</th>
        <th>Naam</th>
        <th>Prijs</th>
        <th>Omschrijving</th>
    </TableHeader>
    <RowTemplate>
        <td>@context.Id</td>
        <td>@context.Naam</td>
```

```
            <td>@context.Prijs</td>
            <td>@context.Omschrijving</td>
        </RowTemplate>
        <TableFooter>
            &copy; VDAB 2020
        </TableFooter>
    </TableTemplate>

    @code {
        public IEnumerable<Artikel> artikels { get; set; } = new List<Artikel>() {
            new Artikel {Id = 1, Naam = "Zaag", Prijs = 11.50m,
                         Omschrijving = "Handzaag 50cm" },
            new Artikel { Id = 2, Naam = "Hamer", Prijs = 74.99m,
                          Omschrijving = "Voorhamer 10kg" },
            new Artikel { Id = 3, Naam = "Beitel", Prijs = 15.75m,
                          Omschrijving = "Steenbeitel 30cm" }
        };
    }
```

De TableTemplate heeft een attribuut *Items.* Die krijgt als waarde de List<Artikel> *artikels* die in de code-section is geïnitialiseerd. De items worden doorgegeven aan de parameter property Items in de template.

Vervolgens wordt de inhoud van de 3 RenderFragments beschreven : in TableHeader worden de kolomnamen doorgegeven, in TableFooter de footertekst. RowTemplate is een RenderFragment<T> waarbij T in dit geval een artikel is. Verwijzen naar het artikel doen we via **@context**.

We missen wel nog een using-statement voor de folder *Models.*

- Voeg in *_Imports.razor* een using-statement toe zodat de weg naar de folder waarin de class Artikel staat wordt gevonden.
- Probeer uit : start de app en kies in het menu de optie *Tables*.

Het resultaat :

Id	Naam	Prijs	Omschrijving
1	Zaag	11,50	Handzaag 50cm
2	Hamer	74,99	Voorhamer 10kg
3	Beitel	15,75	Steenbeitel 30cm

© VDAB 2020

Nog twee opmerkingen.

In ons voorbeeld geven we een List<Artikel> door aan de component. Blazor leidt hieruit af dat TItem dus een Artikel-object is. In sommige gevallen lukt Blazor hier niet in. We kunnen dan in de oproep van de component zelf aangeven wat de naam van het type is voor Titem :

```
<TableTemplate Items="artikels" TItem="Artikel">
    …
</TableTemplate>
```

- Wijzig eventueel in Tables.razor de begintag van de component *TableTemplate* zoals hierboven weergegeven en probeer opnieuw uit.

Tweede opmerking : in de template gebruiken we property's van TItem via *context*, bijvoorbeeld `@context.Id`. Om duidelijk te maken dat context hier een artikel is kunnen we zelf een naam geven aan context via de parameter *Context* :

```
<TableTemplate Items="artikels" TItem="Artikel" Context="artikel">
    …
    <RowTemplate>
        <td>@artikel.Id</td>
        <td>@artikel.Naam</td>
        <td>@artikel.Prijs</td>
        <td>@artikel.Omschrijving</td>
    </RowTemplate>
    …
</TableTemplate>
```

- Probeer opnieuw uit met de wijzigingen zoals hierboven weergegeven.

Om te bewijzen dat we een universele, herbruikbare template gemaakt hebben, gebruiken we de template *TableTemplate* nu voor een personenlijst.

- Voeg eerst onderstaande class *Persoon* toe in de folder *Models*.

```
public class Persoon
{
    public int Id { get; set; }
    public string Voornaam { get; set; }
    public string Familienaam { get; set; }
}
```

- Voeg onderstaande code toe in *Tables.razor* :

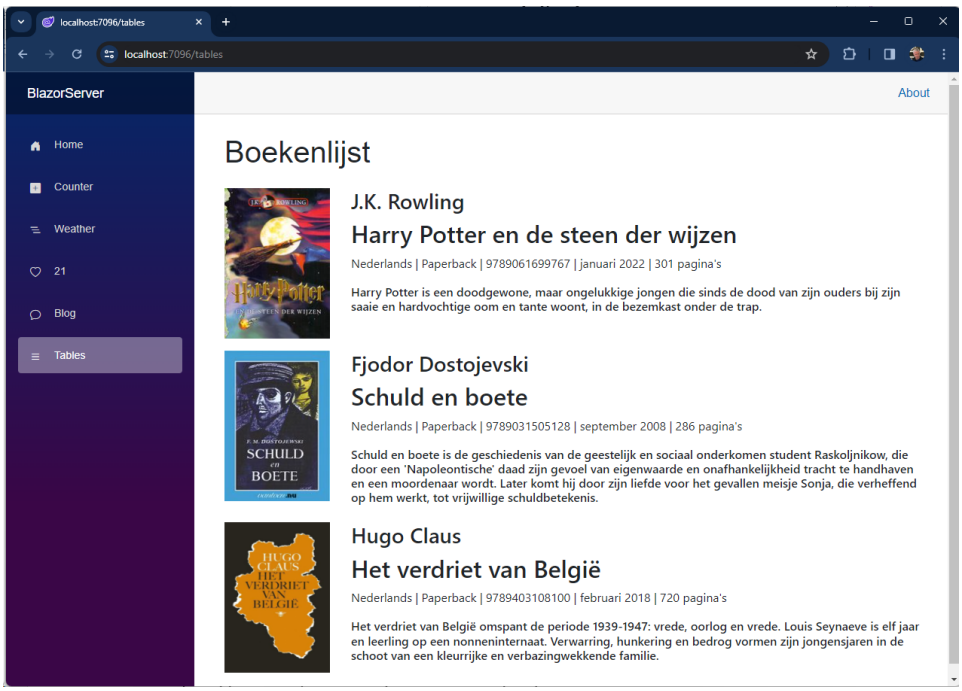
```
…
<TableTemplate Items="personen" TItem="Persoon" Context="Persoon">
    <TableHeader>
        <th>Id</th>
        <th>Voornaam</th>
        <th>Familienaam</th>
    </TableHeader>
    <RowTemplate>
        <td>@Persoon.Id</td>
        <td>@Persoon.Voornaam</td>
        <td>@Persoon.Familienaam</td>
    </RowTemplate>
</TableTemplate>

@code {
    …
    public IEnumerable<Persoon> personen { get; set; } = new List<Persoon>() {
        new Persoon { Id = 1, Voornaam = "Steven", Familienaam = "Lucas" },
        new Persoon { Id = 2, Voornaam = "John", Familienaam = "Fitzgerald" },
        new Persoon { Id = 3, Voornaam = "Rita", Familienaam = "Jones" }
    };
}
```

Om aan te tonen dat je niet verplicht bent alle renderfragments op te nemen hebben we deze keer de tabel geen footer gegeven.

- Probeer opnieuw uit.

3.7.5 Oefening

Gebruik de TableTemplate om onderstaande boekenlijst te tonen.



Je gebruikt de TableHeader om de titel *Boekenlijst* af te beelden. In een RowTemplate beeld je de gegevens van een boek af. Een boek heeft de property's *Auteur, Titel, Afbeelding, Taal, Afwerking, ISBN, UitgifteDatum, AantalPaginas* en *Beschrijving.*

3.7.6 Attribute splatting

In een vorige paragraaf hebben we een CopyrightFooter component gebruikt in de component *Eenentwintig*. Aan die CopyrightFooter component gaven we een waarde mee voor de auteur via een attribuut Auteur en ook een childcontent.

We vinden dit onvoldoende en we willen ook nog een jaartal, de naam van een cursus en een doelgroep meegeven. Dit betekent dus dat we extra parameter property's moeten toevoegen in de CopyrightFooter component en de layout moeten aanpassen.

- Wijzig de code in *CopyrightFooter.razor* als volgt :

```
<div class="alert alert-info">
    <div>@Doelgroep - @Cursus</div>
    <div>&copy; @Auteur (@Jaartal)</div>
    <div>@ChildContent</div>
</div>

@code {

    [Parameter]
    public string? Auteur { get; set; }

    [Parameter]
    public string? Cursus { get; set; }

    [Parameter]
    public int Jaartal { get; set; }
```

```
        [Parameter]
        public string? Doelgroep { get; set; }

        [Parameter]
        public RenderFragment? ChildContent { get; set; }

    }
```

Wanneer we nu in de component Eenentwintig de component CopyrightFooter willen gebruiken, dan moeten we deze parameterproperty's een waarde geven via attributen :

```
<CopyrightFooter Auteur="Steven" Cursus="Blazor" Jaartal="2020"
   Doelgroep=".NET ontwikkeling met C#">
    Blazor rocks
</CopyrightFooter>
```

Deze werkwijze heeft twee nadelen : ten eerste is het veel tikwerk. Voor elke parameter moet er een attribuut met bijhorende waarde toegevoegd worden aan de tag `<CopyrightFooter>` .

Ten tweede zijn de waarden die doorgegeven worden aan de copyrightfooter hardcoded strings. Deze kunnen we at runtime via code niet wijzigen. Als we dat willen doen moeten we voor elke parameter een private variabele toevoegen in de component *Eenentwintig*.

We kunnen dit oplossen met een techniek die *attribute splatting* wordt genoemd. We voegen de attributen in één keer toe onder de vorm van een Dictionary<string, object>.

We definiëren in de component *Eenentwintig* een extra veld van het type Dictionary<string, object>. Daaraan voegen we via code een aantal elementen toe bestaande uit de naam van het attribuut en de waarde. Deze dictionary is dynamisch aanpasbaar.

- Voeg onderstaande code toe in de @code-sectie in de component *Eenentwintig* :

```
@code {
    …
    private Dictionary<string, object> footerAttributes =
        new Dictionary<string, object>() {
            {
              "auteur", "Steven"
            },
            {
              "cursus", "Blazor"
            },
            {
              "jaartal", 2020
            },
            {
              "doelgroep", ".NET ontwikkeling met C#"
            },
        };
    …
}
```

We kunnen deze attributen nu in één keer toepassen op de component *CopyrightFooter* door de dictionary toe te kennen aan een attribuut met de naam `@attributes`.

- Wijzig de layoutcode in de component *Eenentwintig* als volgt :

```
<CopyrightFooter @attributes="@footerAttributes">
    Blazor rocks
</CopyrightFooter>
```

De attributen worden nu in één keer doorgegeven aan de component door de inhoud van het veld *footerAttributes* toe te kennen aan *@attributes.* Dit is minder tikwerk.

- Je kan dit uitproberen door de app te starten en het menuitem *Eenentwintig* te kiezen.

Het resultaat :

.NET ontwikkeling met C# - Blazor
© Steven (2020)
Blazor rocks

In de loop van het programma kan de dictionary ook aangepast worden zodat de waarden van de attributen veranderen. We proberen dit uit.

- Voeg in *Eenentwintig.razor* onderaan de layout een button toe :

```
…
<CopyrightFooter @attributes="@footerAttributes">
    Blazor rocks
</CopyrightFooter>

<button class="btn btn-primary m-3" @onclick="Verander">
    Verander auteur
</button>
```

- Voeg in de code section een method *Verander()* toe :

```
@code {
    …
    private void Verander() => footerAttributes["auteur"] = "Jantje";
}
```

- Probeer terug uit. De naam van de auteur is nu dynamisch : een klik op de knop verandert de naam van de auteur in de component.

3.8 Refereren naar een childcomponent

Wanneer je een publieke member van een childcomponent wil aanspreken vanuit het moeder-component dan moet je op één of andere manier naar het childcomponent kunnen refereren. Dit kan door het childcomponent te voorzien van een @ref-attribuut en een bijhorende waarde.

- Wijzig in de component *Eenentwintig* de code voor de CopyrightFooter als volgt :

```
<CopyrightFooter @attributes="@footerAttributes" @ref="cfdynamic">
    Blazor rocks
</CopyrightFooter>
```

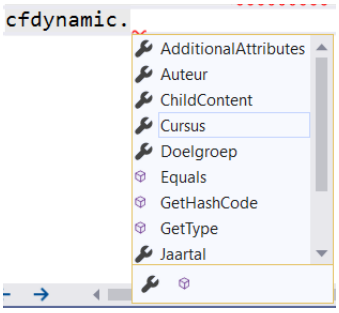
De CopyrightFooter-component heeft nu een referentienaam *cfdynamic.* Deze naam kies je zelf en moet uiteraard verschillen van andere referenties.

Om naar deze childcomponent te kunnen verwijzen moeten we in de code-sectie ook een bijkomende member toevoegen met dezelfde naam als de referentienaam van de component. Het type van dit member is gelijk aan de naam van de childcomponent.

- Voeg in de component *Eenentwintig* onderstaande code toe in de @code-sectie:

```
@code {
    …
    private CopyrightFooter? cfdynamic;
    …
}
```

We kunnen de publieke members van de childcomponent met referentienaam cfdynamic nu aanspreken vanuit de code van de moedercomponent :



We proberen dit uit. Vanuit de component *Eenentwintig* vragen we de cursusnaam uit de childcomponent *CopyrightFooter* op én roepen we een public method uit deze childcomponent op.

We starten met het toevoegen van een method in de component *CopyrightFooter*. Die starten we zo meteen op vanuit de moedercomponent.

- Voeg onderstaande method toe in de code-sectie van de component *CopyrightFooter* :

```
public string AlleInfo()
{
    return $"{Auteur} {Cursus} {Jaartal} {Doelgroep}";
}
```

We voegen nu in de moedercomponent een knop toe. Een klik op deze knop zal de method uit de childcomponent oproepen.

- Voeg in de component *Eenentwintig* onder de component *CopyrightFooter* onderstaande code toe :

```
<button class="btn btn-primary m-3" @onclick="InfoOphalen">Info</button>
<div>@cursus</div>
<div>@info</div>
```

- In de code-sectie van de component *Eenentwintig* voeg je onderstaande code toe :

```
private string? cursus;
private string? info;

private void InfoOphalen()
{
    cursus = cfdynamic?.Cursus;
    info = cfdynamic?.AlleInfo();
}
```

In de method *InfoOphalen()* kunnen we de publieke members van de childcomponent aanspreken via de referentienaam van de childcomponent : *cfdynamic*. Het resultaat stoppen we in de membervariabelen van de moedercomponent.

- Probeer uit : klik op de knop *Info* en je krijgt de informatie uit de childcomponent te zien.

3.9 Code in een aparte class opnemen

Onze componenten bevatten nu zowel layoutcode, uitgedrukt in razor, als C#-code in een @code-sectie. We kunnen de layout- en de C#-code ook van elkaar scheiden door ze in aparte bestanden onder te brengen.

Daarvoor zijn twee mogelijke werkwijzen : ofwel slaan we de C#-code op in een partial class met extensie .razor.cs, ofwel stoppen we de C#-code in een zogenaamde base class.

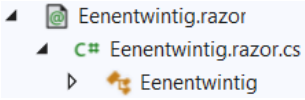
3.9.1 Code in een partial class

We kunnen (liefst alle) code uit de @code-sectie van een razor component in een aparte class onderbrengen. Deze class is een partial class met dezelfde naam als de razor component met daaraan de extensie .cs toegevoegd.

We proberen dit uit met een component.

- Voeg in de folder *Pages* een nieuwe class toe met de naam *Eenentwintig.razor.cs*.
- Maak van deze class een partial class door het keyword *partial* toe te voegen.

Je ziet in de Solution Explorer dat deze nieuwe class een onderdeel wordt van de component *Eenentwintig.razor* :



- Verplaats nu alle code uit de @code-sectie (dus alles wat er tussen de accolades na `@code` staat) van *Eenentwintig.razor* naar deze nieuwe class.
- Probeer opnieuw uit : start de app en kies het menuitem *Eenentwintig.*

Op zich kan je een deel van de C#-code laten staan in de @code-sectie en een ander deel in de partial class onderbrengen maar dit raden we niet aan omwille van de leesbaarheid van de code.

3.9.2 Code in een baseclass

In plaats van te werken met partial classes kan je de razor component ook laten erven van een ComponentBase-class.

- Hernoem in de Solution Explorer het partial class bestand *Eenentwintig.razor.cs* naar *EenentwintigBase.cs*.
- Verwijder in de class het keyword *partial,* hernoem de class naar *EenentwintigBase* en laat de class ervan van ComponentBase :

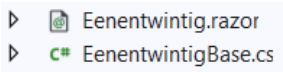
  ```
  public class EenentwintigBase : ComponentBase
  ```

- Voeg in *Eenentwintig.razor* net onder de @page-directive onderstaande code toe :

  ```
  @inherits EenentwintigBase
  ```

- Wijzig tenslotte in *EenentwintigBase.cs* het protection level van de members die in *Eenentwintig.razor* worden aangesproken van private in (minstens) protected.
- Je kan opnieuw uitproberen.

Deze werkwijze heeft wel als nadeel dat het overzicht in de Solution Explorer al snel verloren gaat omdat het razor-bestand en de bijhorende baseclass niet worden gegroepeerd :



3.10 Oefening

Splits de layout en de C#-code van de component *Dobbelen* op in aparte bestanden. Kies zelf welke methode je daarvoor gebruikt.

3.11 Cascading values

In één van de voorbije paragrafen heb je geleerd dat een component een waarde kan doorgeven aan zijn childcomponent via een component parameter. De childcomponent kan dan op zijn beurt de waarde verder doorgeven aan zijn childcomponent en zo verder.

Dit vergt heel veel codeerwerk voor het toevoegen van de component parameters, ook in componenten uit de ketting die de waarde niet zelf nodig hebben en enkel maar moeten doorgeven.

We lossen dit probleem op met cascading values.

3.11.1 Voorbeeld met component parameters

Vooraleer we een elegante oplossing aanreiken met cascading values maken we een voorbeeld waarbij we de gekende component parameters gebruiken.

- Maak in de folder *Pages* een nieuwe razor component met de naam *Cascade.razor*.
- Voeg in *NavMenu* een menuitem toe voor deze component :

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="cascade">
        <span class="bi bi-list-nested"
              aria-hidden="true"></span> Cascading Values
    </NavLink>
</div>
```

- Maak in de folder *Components* een razor component *SubComponent.razor*.

De component *Cascade* voorzien we van een member *Getal* dat we op 10 initialiseren. Via een knop kan de gebruiker dit getal verhogen met 1. We geven de waarde van *Getal* door aan *SubComponent* via een parameter *SubGetal*.

- Voorzie de component *Cascade.razor* van volgende code :

```
@page "/cascade"
<h1>Cascade</h1>
<p>Waarde getal in cascade.razor : @Getal</p>
<button class="btn btn-primary mb-2"
    @onclick="Verhoog">Verhoog de waarde van getal met 1</button>

<SubComponent SubGetal="@Getal" />

@code {
    private int Getal = 10;
```

```
        private void Verhoog()
        {
            Getal++;
        }
    }
```

In *SubComponent* geven we nu de waarde van het doorgegeven getal weer.

- Wijzig de code in *SubComponent.razor* als volgt :

```
<div class="ms-4 mt-4" style="background-color:lightblue;padding:30px;">
    <h3>SubComponent</h3>
    <p>Doorgegeven waarde : @SubGetal</p>
</div>

@code {
    [Parameter]
    public int SubGetal{ get; set; }
}
```

- Probeer uit. Dit is het resultaat na enkele keren klikken op de knop :

We voegen nu een extra niveau toe door een component *SubSubComponent* op te nemen in *SubComponent*.

- Voeg een razor component *SubSubComponent* toe in de folder *Components*.
- Voorzie deze component van onderstaande code :

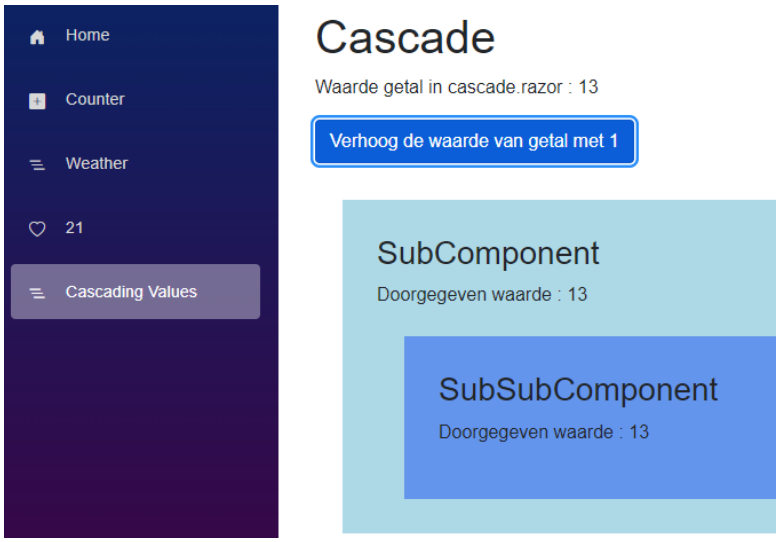
```
<div class="ms-4 mt-4" style="background-color:cornflowerblue;padding:30px;">
    <h3>SubSubComponent</h3>
    <p>Doorgegeven waarde : @SubSubGetal</p>
</div>

@code {
    [Parameter]
    public int SubSubGetal { get; set; }
}
```

Deze component krijgt de waarde vanuit SubComponent binnen via zijn parameter *SubSubGetal*. We voegen deze component nu toe aan SubComponent.

- Wijzig de code in *SubComponent.razor* als volgt :

```
<div class="ms-4 mt-4" style="background-color:lightblue;padding:30px;">
    <h3>SubComponent</h3>
    <p>Doorgegeven waarde : @SubGetal</p>
    <SubSubComponent SubSubGetal="@SubGetal"/>
</div>
```

- Probeer opnieuw uit en klik enkele keren op de button. Het getal uit *Cascade.razor* wordt nu op drie verschillende niveaus weergegeven :



3.11.2 Cascading values i.p.v. component parameters

We vervangen de component parameters nu door zogenaamde *Cascading Values*.

We wrappen in *Cascade.razor* de oproep van de childcomponent *SubComponent* in een CascadingValue-component. De parameter *SubGetal* mag weg in de oproep van het childcomponent.

- Wijzig de code in *Cascade.razor* als volgt :

```
…
</button>
<CascadingValue Name="HetGetal" Value="@Getal">
    <SubComponent />
</CascadingValue>

@code {
…
```

Om in de component *SubComponent* de waarde van de cascading value *HetGetal* op te vragen vervang je in de subcomponent de Parameter door een CascadingParameter.

- Wijzig de @code-sectie van *SubComponent.razor* als volgt :

```
@code {
    [CascadingParameter(Name = "HetGetal")]
    public int SubGetal { get; set; }
}
```

We doen hetzelfde met SubSubComponent. Ook daar vervangen we de Parameter door een CascadingParameter.

- Wijzig ook in *SubSubComponent* de @code sectie :

```
@code {
    [CascadingParameter(Name = "HetGetal")]
    public int SubSubGetal { get; set; }
}
```

- In *SubComponent.razor* verwijder je tenslotte de parameter in de oproep van SubSubComponent:

```
<SubSubComponent />
```

- Je kan opnieuw uitproberen. De waarde wordt op alle niveaus correct weergegeven.

Conclusie : veel hebben we niet gewonnen. Er is nauwelijks code weg uit de childcomponenten en zelfs code toegevoegd in de hoofdcomponent *Cascade.razor*. De winst komt er als we op één of meerdere subniveaus de cascading value niét nodig hebben. We gaan de cascading parameter in de middelste component, *SubComponent*, verwijderen en toch zal de waarde in *SubSubComponent* perfect weergegeven worden. Dit zou met gewone parameters niet mogelijk zijn. Daar moet een waarde van niveau tot niveau doorgegeven worden.

We passen de code in *SubComponent* aan zodat de cascading value niet wordt weergegeven. In de codesection staat nu geen code meer. Deze component speelt nu geen rol meer in het doorgeven van de waarde.

- Wijzig *SubComponent.razor* als volgt :

```
<div class="ms-4 mt-4" style="background-color:lightblue;padding:30px;">
    <h3>SubComponent</h3>
    <p>Doorgegeven waarde : @SubGetal</p>
    <SubSubComponent />
</div>

@code {
    [CascadingParameter(Name = "HetGetal")]
    public int SubGetal { get; set; }
}
```

- Probeer opnieuw uit. De waarde van het getal wordt perfect doorgegeven van Cascade naar SubSubComponent via SubComponent, ook al is hiervoor in SubComponent geen code geschreven.

3.11.3 Meerdere cascading values op hetzelfde niveau

In ons voorbeeld definieerden we één cascading value, *HetGetal*, in *Cascade.razor*. We kunnen meerdere cascading values beschikbaar stellen door ze genest te noteren.

Als voorbeeld voegen we een tweede cascading value toe in *Cascade.razor*.

- Wijzig de code in *Cascade.razor* als volgt :

```
…
<CascadingValue Name="HetGetal" Value="@Getal">
    <CascadingValue Name="HetWoord" Value="@Woord">
        <SubComponent />
    </CascadingValue>
</CascadingValue>

@code {
    private int Getal = 10;
    private string Woord = "VDAB";
    …
}
```

Opmerking

In bovenstaand voorbeeld maakt de volgorde waarin de twee cascading values in elkaar zijn genest niet uit. Beide cascading values worden doorgegeven aan SubComponent. In onderstaand voorbeeld is de volgorde wél van belang :

```
<CascadingValue Name="HetGetal" Value="@Getal">
    <SubComponent />
    <CascadingValue Name="HetWoord" Value="@Woord">
        <SubComponent />
    </CascadingValue>
</CascadingValue>
```

In dit voorbeeld bevat *Cascade.razor* twee keer de component *SubComponent*. Beide zitten binnen de begin- en eindtag van de cascading value *HetGetal* en kunnen die value dus opvangen en weergeven. Enkel de tweede SubComponent zit ook nog eens tussen de begin- en eindtag van de cascading value *HetWoord* en kan dus *HetWoord* opvangen en weergeven.

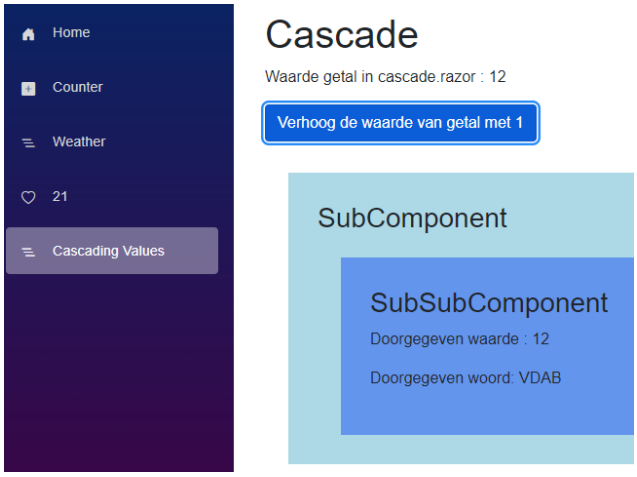
Nu de cascading value *HetWoord* gedefinieerd is kan deze in alle subcomponenten waar nodig opgevangen worden via een CascadingParameter. Als voorbeeld tonen we de inhoud van de cascading value *HetWoord* in *SubSubComponent*.

- Wijzig de code in *SubSubComponent.razor* als volgt :

```
<div class="ms-4 mt-4" style="background-color:cornflowerblue;padding:30px;">
    <h3>SubSubComponent</h3>
    <p>Doorgegeven waarde : @SubSubGetal</p>
    <p>Doorgegeven woord: @DoorgegevenWoord</p>
</div>

@code {
    [CascadingParameter(Name = "HetGetal")]
    public int SubSubGetal { get; set; }
    [CascadingParameter(Name = "HetWoord")]
    public string? DoorgegevenWoord { get; set; }
}
```

- Probeer uit.

Het woord uit de component *Cascade* wordt twee niveaus lager opgevangen en weergegeven :



Wanneer je héél veel cascading values hebt in een component en jouw code stilaan begint te lijken op de Spaanse trappen van Rome dan kan je overwegen om de cascading values te groeperen in een class. Je geeft dan slechts één cascading value door met als type de class.

Code met afzonderlijke cascading values :

```
<CascadingValue Name="Eerste" Value="…">
  <CascadingValue Name="Tweede" Value="…">
    <CascadingValue Name="Derde" Value="…">
      <SubComponent />
    </CascadingValue>
  </CascadingValue>
</CascadingValue>
```

Code met cascading values gegroepeerd in een class :

```
<CascadingValue Name="Gegevens" Value="@Gegevens">
    <SubComponent />
</CascadingValue>
```

Hier is *Gegevens* een member van een type met bijvoorbeeld volgende classbeschrijving :

```
public class GegevensClass
{
    public int Eerste { get; set; }
    public string? Tweede { get; set; }
    public string? Derde { get; set; }
}
```

We proberen dit uit.

- Voeg bovenstaande class *Gegevens* toe in de folder Models.
- Voeg in de component *Cascade.razor* een extra property toe :

```
@code {
    …
    private Gegevens Gegevens = new Gegevens {
        Eerste = 1, Tweede = "Twee", Derde = "Derde"
    };
    …
}
```

- In de layout van de component *Cascade.razor* voeg je onderaan onderstaande code toe :

```
<CascadingValue Name="Gegevens" Value="@Gegevens">
   <SubComponent />
</CascadingValue>
```

Bij wijze van voorbeeld zullen we nu de gegevens weergeven in de component SubSubComponent.

- Voeg in SubSubComponent.razor in de codesectie een cascading parameter toe :

```
…
[CascadingParameter(Name = "Gegevens")]
public Gegevens? Gegevens { get; set; }
…
```

- Vul de layoutcode in SubSubComponent.razor als volgt aan :

```
<div class="ms-4 mt-4" style="background-color:cornflowerblue;padding:30px;">
    …
    <p>Doorgegeven gegevens : @Gegevens?.Eerste, @Gegevens?.Tweede,
@Gegevens?.Derde</p>
</div>
```

- Je kan opnieuw uitproberen.

3.11.4 Meerdere cascading values op verschillende niveaus

In ons voorbeeld werden de cascading values gedefinieerd op het hoogste niveau in *Cascade.razor*. Je kan ook cascading values definiëren op een subniveau.

Onderstaande code maakt een cascading value *GetalVanSubComponent* beschikbaar in alle child-componenten van *SubComponent.razor*. De cascading value is deze keer bij wijze van voorbeeld de constante waarde 13.

- Wijzig de code in *SubComponent.razor* als volgt :

```
<div class="ms-4 mt-4" style="background-color:lightblue;padding:30px;">
    <h3>SubComponent</h3>
    <CascadingValue Name="GetalVanSubComponent" Value="13">
        <SubSubComponent />
    </CascadingValue>
</div>
```

In SubSubComponent kan de cascading value opgevangen worden via een cascadingparameter.

- Wijzig de code in *SubSubComponent.razor* als volgt :

```
<div class="ms-4 mt-4" style="background-color:cornflowerblue;padding:30px;">
    <h3>SubSubComponent</h3>
    …
    <p>Getal van subcomponent: @SubGetal</p>
</div>

@code {
    …
    [CascadingParameter(Name = "GetalVanSubComponent")]
    public int SubGetal { get; set; }
}
```

- Probeer uit. Je ziet in de subsubcomponent de waarde 13 uit subcomponent.

3.11.5 Cascading value by type

In alle bovenstaande voorbeelden werd de cascading value geïdentificeerd door een naam. Via deze naam kon de cascading value opgevangen worden in de cascading parameter.

Je kan deze naam ook weglaten. De koppeling gebeurt dan niet op de naam van de cascading value maar op het type. Maar wat als je meerdere cascading values hebt met hetzelfde type? Welke wordt dan opgevangen? Het antwoord is : de cascading value van het gewenste type die het dichtstbij is gedefinieerd.

Als voorbeeld definiëren we in *SubComponent.razor* een extra cascading value (zonder naam) van het type string. Vervolgens vangen we in SubSubComponent een cascading value op van het type string.

Het is de waarde uit SubComponent (en niet die uit Cascade) die wordt opgevangen omdat deze dichterbij gedefinieerd is.

- Wijzig de code in *SubComponent.razor* als volgt :

```
<div class="ms-4 mt-4" style="background-color:lightblue;padding:30px;">
    <h3>SubComponent</h3>
    <CascadingValue Name="GetalVanSubComponent" Value="13">
        <CascadingValue Value="@woord" >
            <SubSubComponent />
        </CascadingValue>
    </CascadingValue>
</div>

@code {
    private string woord = "Dichterbij";
}
```

- Verwijder nu in *Cascade.razor* de naam van de cascading value van het type string :

```
…
<CascadingValue Name="HetGetal" Value="@Getal">
    <CascadingValue Name="HetWoord" Value="@Woord">
        <SubComponent />
    </CascadingValue>
</CascadingValue>
…
```

- Wis in SubSubComponent de parameter Name in de annotation bij DoorgegevenWoord.

```
…
[CascadingParameter]
public string? DoorgegevenWoord { get; set; }
…
```

- Probeer opnieuw uit. De waarde die wordt weergegeven in SubSubComponent is de dichtstbijzijnde en dat is “Dichterbij” en niet “VDAB” :

3.11.6 Vaste cascading values

Wanneer we de waarde van een cascading value veranderen dan verandert de waarde in de subcomponenten mee. Dit vergt heel wat werk van Blazor. Wanneer we zeker weten dat de cascading value na het doorgeven niet meer zal veranderen dan kunnen we dat aangeven met een attribuut *IsFixed*. Blazor hoeft dan niet continu te checken of de cascading value is veranderd.

In de component *SubComponent* geven we het getal 13 door. Dit is een vast getal en zal niet veranderen. Je kan Blazor heel wat onnodig werk besparen door een attribuut *IsFixed* toe te voegen met de waarde *true*.

De code in *SubComponent.razor* wordt dan :

```
<CascadingValue Name="GetalVanSubComponent" Value="13" IsFixed="true">
```

3.11.7 Cascading values overriden

Je kan de waarde van een hogerop in de componentenhiërarchie gedefinieerde cascading value overriden door de cascading value in één van de subcomponenten opnieuw te definiëren. Alle code **tussen begin- en eindtag van de nieuwe cascading value definitie** zal de nieuwe waarde bevatten.

Als voorbeeld voegen we in SubComponent een tweede keer de component SubSubComponent toe. De tweede plaatsen we binnen een nieuwe definitie van de cascading value *HetGetal*. Deze overschrijft de gelijknamige cascading value uit de component *Cascade.*

- Wijzig de code in *SubComponent.razor* als volgt :

```
…
<CascadingValue Name="GetalVanSubComponent" Value="13" IsFixed="true">
    <CascadingValue Value="@woord">
        <SubSubComponent />
        <CascadingValue Name="HetGetal" Value="1">
            <SubSubComponent/>
        </CascadingValue>
    </CascadingValue>
</CascadingValue>
…
```

- Probeer uit. In de eerste SubSubComponent wordt de waarde uit Cascade weergegeven en in de tweede SubSubComponent de nieuwe waarde 1 (en niet de waarde uit Cascade.razor).

3.12 Oefening

Definieer in de component *Cascade.razor* een cascading value van het type *Persoon.* Vang de waarde op in *SubSubComponent* en geef ze daar weer.

4 Data binding

Wanneer we de waarde van een member weergeven in een page doen we eigenlijk aan one-way data binding. Wordt de waarde via code gewijzigd dan past de weergave zich automatisch aan. Soms zullen we de afgebeelde waarde ook willen veranderen vanuit de UI, dus in de omgekeerde richting. We spreken dan van two-way binding.

4.1 Two-way binding

We gebruiken het attribuut *@bind* om de value van een inputelement te binden aan een componentmember.

⇒ We geven de waarde van een member weer in een textbox. Verandert de waarde van de member door een wijziging in de code dan wordt de nieuwe waarde weergegeven in de textbox.

⇐ Passen we de waarde in de textbox aan dan wordt de waarde van de member aangepast bij het verlaten van de textbox.

Als voorbeeld maken we een page waarop we de rode, groene en blauwe component van een kleur kunnen instellen. De bijhorende kleur wordt eronder weergegeven alsook de hexadecimale waarde.

- Maak in de folder *Pages* een nieuwe Razor component en noem deze *Kleur.razor*.
- Voeg in NavMenu een verwijzing toe naar deze component :

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="kleur">
        <span class="bi bi-brush" aria-hidden="true">
        </span> Kleur
    </NavLink>
</div>
```

- Voeg onderstaande code toe aan de component *Kleur* :

```
@page "/kleur"
<h1>Kleur</h1>
<div class="mb-3 row mt-3">
    <label for="r" class="col-sm-1 col-form-label">Rood</label>
    <div class="col-sm-1">
        <input id="r" @bind="r" class="form-control" />
    </div>
</div>
<div class="mb-3 row">
    <label for="g" class="col-sm-1 col-form-label">Groen</label>
    <div class="col-sm-1">
        <input id="g" @bind="g" class="form-control" />
    </div>
</div>
<div class="mb-3 row">
    <label for="b" class="col-sm-1 col-form-label">Blauw</label>
    <div class="col-sm-1">
        <input id="b" @bind="b" class="form-control" />
    </div>
</div>
<div class="form-group row mt-3">
    <div class="col-sm-4 ms-3" style=
        "background-color:rgb(@r,@g,@b);height:50px;border-radius: 10px;"/>
    <label class="col-sm-1 col-form-label">#@HexColor()</label>
</div>
```

```
@code {
    private byte r = 0;
    private byte g = 0;
    private byte b = 0;

    private string HexColor()
    {
        return (r.ToString("X2") + g.ToString("X2") + b.ToString("X2"));
    }
}
```

Bij aanvang zorgen de beginwaarden van de members r, g en b voor een zwarte kleur. De input-velden geven de waarde 0 weer en de hexadecimale waarde #000000 wordt getoond. Na het aanpassen van de inputvelden veranderen de members r, g en b en dus ook de kleur en de hexadecimale weergave.

- Probeer uit.

4.2 Bind-event

In ons programma wordt de binding pas doorgevoerd wanneer een inputveld de focus verliest. Op dat moment treedt het onchange event op. Je kan de binding ook laten gebeuren bij een ander event zoals na het intikken van een teken (oninput event). De gebruiker krijgt dan sneller feedback.

Je gebruikt @bind:event="*naamvanhetevent*" om expliciet aan te geven bij welk event de binding moet gebeuren.

- Voeg bij elk inputveld de code `@bind:event="oninput"` toe.

```
…
<input id="r" @bind="r" @bind:event="oninput" class="form-control" />
…
```

- Probeer uit. De kleur en de hexadecimale waarde verandert nog voor je het veld verlaat.

4.3 Ongeldige waarden

Wanneer we een ongeldige waarde invullen voor één van de kleurcomponenten dan wordt de oude waarde automatisch hersteld. We zien dit effect het duidelijkst wanneer we kiezen voor het onchange event.

- Verwijder in de code voor de inputvelden terug de code `@bind:event="oninput"`, zodat de defaultevent *onchange* terug wordt gebruikt.
- Start de app en kies in het menu voor de page *Kleur.*

- Vul een geldige waarde in voor rood, bijvoorbeeld 125 en plaats de cursor in het veld voor groen. De kleur wordt aangepast.
- Wijzig de waarde voor rood in een ongeldige waarde, bijvoorbeeld *abc* en plaats de cursor in een ander veld. De oude waarde, 125, wordt hersteld.

We kunnen in ons voorbeeld foutieve input voorkomen door een type in te stellen voor het inputveld en eventueel een minimum- en/of maximumwaarde.

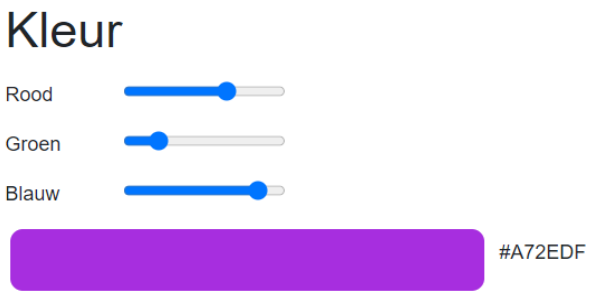
- Voeg aan de inputvelden de code `type="number" min="0" max="255"` toe.

In de meeste browsers krijgen de inputvelden hierdoor aan de rechterkant pijltjes waarmee je de waarde kan veranderen maar niet lager dan 0 of hoger dan 255.

Tik je een waarde buiten het bereik 0-255 of tekst in het veld dan krijgt het inputveld de waarde 0.

Wil je zeker zijn dat je een geldige ingave krijgt, kies dan voor het type *range,* in combinatie met de attributen *min* en *max*. Je krijgt dan een slider die enkel geldige waarden produceert.

- Vervang in de inputvelden het type in *range* en verwijder de class *form-control.*
- Geef de omsluitende div de class `col-sm-3` (i.p.v. `col-sm-1`).
- Verwijder de class col-form-label bij het label.
- Probeer uit.



Opmerking

Met de directive **@bind** geven we aan dat we two-way databinding willen met de default property van een input : value. We kunnen ook expliciet aangeven dat we met deze property willen binden door de directive **@bind-value** te gebruiken.

4.4 Data binding tussen parent en child component via eventcallback

We kunnen makkelijk waarden doorgeven van een parent component naar zijn child componenten. Maar af en toe zullen we ook in de omgekeerde richting, van child naar parent, waarden willen doorgeven. Dit is bijvoorbeeld het geval wanneer de child component een waarde doorkrijgt van de parent, deze waarde wijzigt en de parent hiervan op de hoogte moet gehouden worden.

Een voorbeeld : we maken een parent component waarin we de waarde van een getal via een knop kunnen verhogen met 5. Deze waarde wordt permanent doorgegeven aan een child component dat

dit getal eveneens weergeeft. In de child component kunnen we de waarde via een knop verhogen met 1. Deze wijziging moet zichtbaar zijn in de parent component.

- Maak in de folder *Pages* een nieuwe Razor component en noem deze *Parent.razor*.
- Voeg in NavMenu een verwijzing toe naar deze component :

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="parent">
        <span class="bi bi-box-arrow-up-left" aria-hidden="true">
        </span> Data binding
    </NavLink>
</div>
```

In de component *Parent* voegen we een membervariabele *getal* toe. Dit getal tonen we in de component. Door te klikken op een knop *Verhoog met 5* verhogen we het getal met 5.

- Neem onderstaande code op in *Parent.razor* :

```
@page "/parent"
<h1>Parent</h1>
<p>Getal : @getal</p>
<button class="btn btn-primary" @onclick="Verhoog">Verhoog met 5</button>
@code {
    private int getal;
    private void Verhoog()
    {
        getal+=5;
    }
}
```

We voegen in de folder *Components* een tweede component toe met de naam *Child*. Deze component heeft een parameterproperty die de waarde van Getal meekrijgt vanuit zijn parentcomponent. In de child component kunnen we het getal via een button met 1 verhogen.

- Voeg in de folder *Components* een nieuwe razor component toe en noem deze *Child.razor*.
- Voeg onderstaande code toe :

```
<div style="background-color:aliceblue;padding:20px;margin:20px;">
    <h3>Child</h3>
    <p>Getal : @GetalInChild</p>
    <button class="btn btn-primary" @onclick="VerhoogGetalInChild">
        Verhoog met 1
    </button>
</div>
@code {

    [Parameter]
    public int GetalInChild { get; set; }

    private void VerhoogGetalInChild()
    {
        GetalInChild++;
    }
}
```

Opmerking : voor de duidelijkheid hebben we in de child component de property die het getal bijhoudt, *GetalInChild,* een andere naam gegeven dan in de parent component. Dit is niet per se nodig, je mag ze ook allebei *Getal* noemen.

In *Parent* voegen we nu het childcomponent toe. We geven de waarde van het getal uit de parentcomponent mee als parameter.

- Voeg in *Parent.razor* net onder de button de childcomponent toe :

```
…
<button class="btn btn-primary" @onclick="Verhoog">Verhoog met 5</button>
<Child GetalInChild="@getal" />

@code {
…
```

- Probeer uit : wanneer we in de parent op de button klikken verhoogt het getal zowel in parent als in child met 5. Wanneer we echter in de childcomponent op de button klikken verhoogt het getal in parent niet mee.

We passen dit aan.

- Wijzig in de parent de oproep van de childcomponent als volgt :

```
<Child @bind-GetalInChild="getal"/>
```

Je voegt dus voor de naam van de parameter `@bind-` toe. Opgepast : vlak voor `getal` staat geen `@`-teken meer !

De parent component is nu klaar om een update te ontvangen. In de childcomponent voegen we een mechanisme toe dat de aangepaste waarde doorgeeft aan de parent.

We voegen in de codesectie van *Child.razor* een property toe van het type EventCallback<T> :

```
[Parameter]
public EventCallback<int> GetalInChildChanged { get; set; }
```

Het type dat de eventcallback doorgeeft is een int dus is het type van bovenstaande property een EventCallback<int>. De naam van de eventcallback-property mag je niet vrij kiezen. Deze bestaat uit de naam van de gewijzigde property die je wil doorgeven plus *Changed.* Dus in ons geval wordt dit *GetalInChildChanged*. Je decoreert de eventcallback met een [Parameter]-annotation.

In de method die het getal verhoogt, VerhoogGetalInChild(), roep je de eventcallback op. Dit gebeurt met de asynchrone method *InvokeAsync()*. Het gewijzigde getal wordt als parameter meegegeven. De method *VerhoogGetalInChild()* passen we aan naar een asynchrone method.

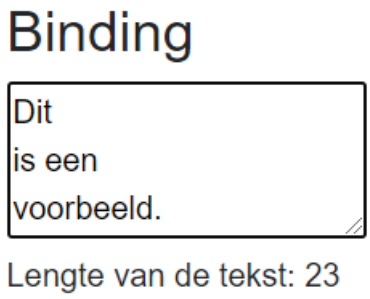
- Wijzig de code in *Child.razor* als volgt :

```
…
@code {

    [Parameter]
    public int GetalInChild { get; set; }

    [Parameter]
    public EventCallback<int> GetalInChildChanged { get; set; }

    private async Task VerhoogGetalInChild()
    {
        GetalInChild++;
        await GetalInChildChanged.InvokeAsync(GetalInChild);
    }
}
```

- Probeer uit : het getal blijft nu synchroon aangepast worden zowel bij verhoging in child als in parent.

4.5 Oefening

Maak een pagina met daarop een textarea. Onder de textarea wordt de lengte van de input in de textarea weergegeven. Zorg ervoor dat de lengte telkens wordt geüpdatet bij elke letter die in de textarea wordt getikt.



5 Event handling

5.1 DOM events afhandelen

In Blazor kunnen we een DOM-event (onclick, onmouseover,…) afhandelen door een zogenaamde directive te gebruiken. Die directive begint met een @ gevolgd door *on* en de naam van een event, bijvoorbeeld *@onclick*.

Vervolgens kunnen we op 2 verschillende manieren code koppelen aan het event. Een eerste mogelijkheid is aanduiden welke method de event afhandelt. In onderstaand voorbeeld handelt de method *VerhoogTeller* de event af :

```
…
<button @onclick="VerhoogTeller">Verhoog teller</button>
@code {
    private int teller;
    private void VerhoogTeller()
    {
        teller++;
    }
}
```

We kunnen ook rechtstreeks code toekennen via een lambda expression :

```
<button @onclick="e => { teller++; }">Verhoog teller</button>
```

Wanneer de eventhandler een parameter heeft dan gebruik je de schrijfwijze met de lambda expression.

Naast onclick zijn onmouseover, onkeypress, onchange, onfocus,… veelgebruikte events. Je krijgt makkelijk een volledige lijst te zien door *@on* te tikken en de intellisense te raadplegen.

De eventhandler kan via een *EventArgs* (of afgeleide) parameter extra informatie over de event leveren. We zien dit in het volgende voorbeeld waarbij we de coördinaten van de muiscursor afbeelden.

- Voeg in de folder *Pages* een component *Events.razor* toe.
- In *NavMenu.razor* voeg je een extra menuitem toe:

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="events">
        <span class="bi bi-lightning" aria-hidden="true"></span> Events
    </NavLink>
</div>
```

- Voeg onderstaande code toe aan de component *Events.razor* :

```
@page "/events"

<h1>Events</h1>
<div style="width:400px;height:400px;background-color:lightblue;border-style:solid;"
     @onmousemove="ToonCoord" @onclick="e => counter = e.Detail"></div>
<div style="margin-top:20px;">
    <p>X : @x</p>
    <p>Y : @y</p>
    <p>clicks : @counter</p>
</div>
```

```
@code {
    double x;
    double y;
    double counter;
    private void ToonCoord(MouseEventArgs e)
    {
        x = e.ClientX;
        y = e.ClientY;
    }
}
```

Wanneer de muis boven het blauwe vak beweegt wordt de method *ToonCoord()* uitgevoerd. De muisbeweging veroorzaakt een *MouseMove*-event en dat event heeft een parameter van het type *MouseEventArgs.* De property's *ClientX* en *ClientY* van deze parameter geven de coördinaat van de muiscursor aan t.o.v. de linkerbovenhoek van de page. We geven deze coördinaten weer via de members *x* en *y*.

Een klik op het vak activeert een lambda-expression. Deze expression stopt de waarde van een property *Detail* in de member *counter.* De property *Detail* geeft aan hoeveel keer er (kort na elkaar) geklikt is met de muis. Op deze manier kunnen we een klik onderscheiden van een dubbelklik.

- Probeer uit.

5.2 Default actions tegenhouden

Je kan de normale werking van een event opheffen en zelf behandelen. Je doet dit door een extra directive toe te voegen met de naam *@onEVENTNAAM:preventDefault*. Je kan het event dan zelf afhandelen in een eigen method. We demonstreren dit in onderstaand voorbeeld waarbij we een *onkeypress*-event opvangen en het ingetikte teken enkel op het scherm laten verschijnen als het een klinker is.

- Voeg in *Events.razor* onderstaande code toe :

```
…
<hr />
Klinker :
<input @bind="input" @onkeypress="CheckInput" @onkeypress:preventDefault
  size="1"/>

@code {

    …
    string? input;
    private void CheckInput(KeyboardEventArgs e) {
        string toets = e.Key;
        switch (toets) {
            case "a":
            case "e":
            case "i":
            case "o":
            case "u":
            case "y":
                input = toets;
                break;
        }
    }
}
```

Aan het *onkeypress*-event van een inputveld, gebonden aan de string-member *input,* voegen we een eventhandler *CheckInput* toe. Een *onkeypress*-event toont normaal het ingetikte teken in het inputveld maar de directive *@onkeypress:preventDefault* houdt dit tegen. In *CheckInput* controleren we welke toets werd ingedrukt. Dit kan via de parameter-property *e.Key.* Is dit een klinker dan kennen we deze toe aan de string *input.*

- Probeer uit. Wanneer je een klinker intikt dan is die zichtbaar in het inputveld, een medeklinker niet.

5.3 Stop eventpropagation

Wanneer we klikken op een genest element, bijvoorbeeld een div binnen een andere div, dan klikken we indirect ook op het parent element. We noemen dit de *propagation* van het click-event.

In onderstaand voorbeeld nesten we 3 -elementen. Klikken op de binnenste div activeert het click-event van de binnenste div en daarna dit van de middenste en tenslotte het clickevent van de buitenste div.

- Voeg onderstaande code toe in *Events.razor* :

```
…
<hr />
<div @onclick="BuitensteDiv" class="mt-4"
     style="padding:30px;background-color:cornflowerblue;border-style:solid;" >
    <h1>Buitenste div</h1>
    <div @onclick="MiddensteDiv" class="m-3"
         style="padding:30px;background-color:lightblue;border-style:solid;" >
        <h2>Middenste div</h2>
        <div @onclick="BinnensteDiv" class="m-3"
             style="padding:30px;background-color:azure;border-style:solid;" >
            <h3>Binnenste div</h3>
        </div>
    </div>
</div>
<div class="mt-3 mb-3">
    <h5>Volgorde : @propagation</h5>
    <button class="btn btn-primary ml-2" @onclick="Wis">Wis output</button>
</div>

@code {
    …
    private string? propagation;
    private void BuitensteDiv() { propagation += "buitenste "; }
    private void MiddensteDiv() { propagation += "middenste "; }
    private void BinnensteDiv() { propagation += "binnenste "; }
    private void Wis() { propagation = ""; }
}
```

We tonen dus 3 geneste div-elementen. Elke div heeft een click-eventhandler die aangeeft dat er werd geklikt op de div. Wanneer je op de binnenste div klikt zal je zien dat zowel *binnenste, middenste* als *buitenste* in de uitvoer verschijnt.

- Probeer uit : klik op de verschillende div-elementen en bekijk de output. Wis eventueel de output met een klik op de button *Wis output.*

Je kan de propagation van een event stoppen door `@onclick:stopPropagation` toe te voegen aan de div.

- Wijzig de code van de middenste div :

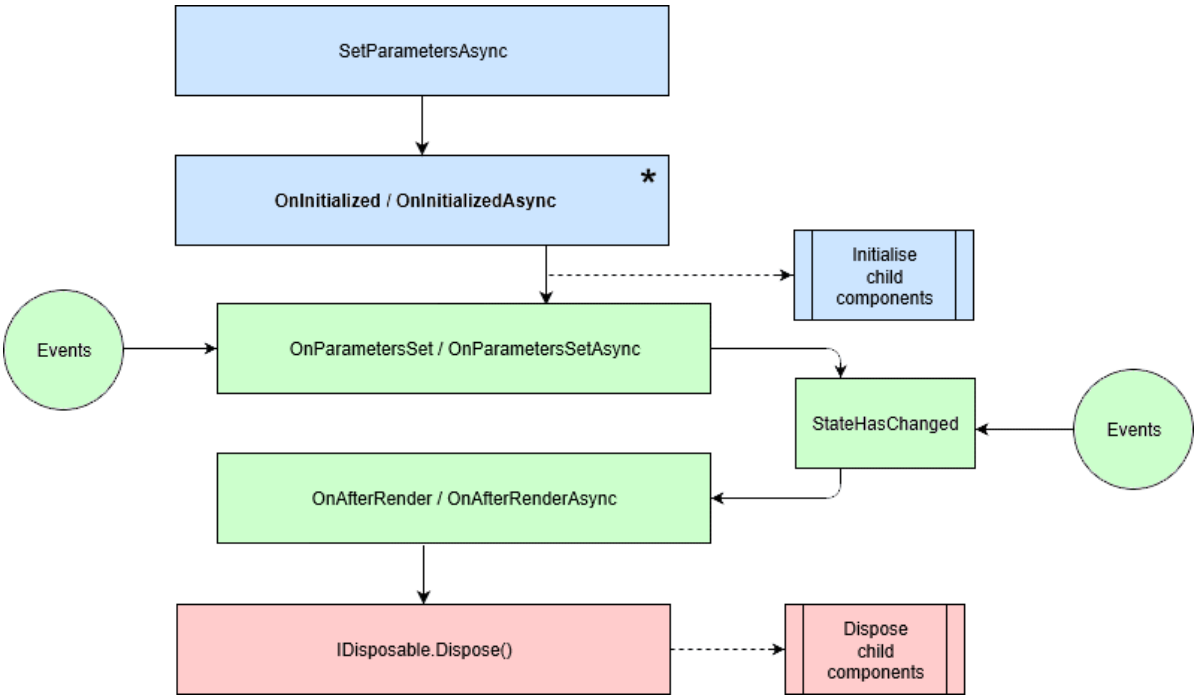
```
<div @onclick="MiddensteDiv" @onclick:stopPropagation …>
```

- Probeer uit. Klik op de binnenste div. De binnenste en middenste div reageren. Het clickevent van de buitenste div wordt niet meer geactiveerd. Het propageren stopt dus bij de middenste.
- Klik op de middenste div. Enkel de middenste div reageert.

6 Component Lifecycles

In dit hoofdstuk staan we even stil bij de lifecycle of levenscyclus van een component. Deze lifecycle start bij de creatie en eindigt met het disposen van de component. Daartussen zijn er een aantal momenten waarop we kunnen ingrijpen en dingen kunnen laten gebeuren. Sommige momenten doen zich één keer voor, andere meerdere keren. Door één of meerdere virtual methods te overriden kunnen we de werking van een component beïnvloeden.

Onderstaande afbeelding geeft de lifecycle en de virtual methods van een component weer.



SetParametersAsync

Bij de oproep van een component kunnen we parameterwaarden meegeven. Achter de schermen wordt een method *SetParametersAsync* opgeroepen en uitgevoerd. In deze method worden de Parameter- en CascadingParameter-property's voorzien van de doorgegeven waarden.

De method *SetParametersAsync* heeft een parameter van het type *ParameterView*. Dit is de collection van alle doorgegeven componentparameters.

Door de method *SetParametersAsync* te overriden kunnen we ingrijpen in het proces van het invullen van de parameters. We kunnen de parameters een defaultwaarde geven, we kunnen ook allerlei controles uitvoeren op de doorgegeven waarden,…

Voorbeeld : in paragraaf 3.6 hebben we in de tweede versie van de dobbelen-oefening de parameter *Aantal* een defaultwaarde 2 gegeven wanneer er geen waarde werd meegegeven via de route. We gaven toen de tip dit te doen in de method *OnInitialized.* Een betere plaats is dus de method *SetParametersAsync* want *Aantal* is een parameter.

- Voeg onderstaande code toe in *Dobbelen.razor.cs* of in *DobbelenBase.cs* zodat de parameter een defaultwaarde krijgt in de method *SetParametersAsync* :

```
public override async Task SetParametersAsync(ParameterView parameters)
{
    await base.SetParametersAsync(parameters);
    if (Aantal == 0) Aantal = 2;
}
```

- Schrap de method *OnInitialized().*
- Probeer opnieuw uit.

OnInitialized/OnInitializedAsync

Na *SetParametersAsync* heeft de component een zekere beginstatus verkregen. Wanneer extra informatie moet worden opgehaald die afhankelijk is van de waarde van een parameter dan kan dit via de method OnInitialized(Async). Een voorbeeld : onze component krijgt de id van een personeels-lid binnen via een parameter. In de method *OnInitialized(Async)* kunnen de personeelsgegevens opgezocht worden in een database.

Let wel : bij een tweede weergave van de component (via een link) wordt de method *OnInitialized* of *OnInitializedAsync* niet meer uitgevoerd. Het wordt wél uitgevoerd als je een nieuwe url intikt in de browser.

We testen dit uit.

- Override in de code van de *Dobbelen* component de method *OnInitialized()* door onderstaande code toe te voegen :

  ```
  protected override void OnInitialized()
  {
      base.OnInitialized();
  }
  ```

- Zet een breakpoint naast de codelijn `base.OnInitialized();`.
- Voeg in de layout van de Dobbelen component onderaan twee links toe :

  ```
  <a href="dobbelen/3">Dobbelen met 3 stenen</a>
  <a href="dobbelen/4">Dobbelen met 4 stenen</a>
  ```

- Start de app mét debugging.
- Browse naar /dobbelen. De debugger houdt een eerste keer halt bij het breakpoint.
- Klik in VS op *Continue.* De debugger stopt een tweede keer aan het breakpoint.
- Klik in VS op *Continue.*
- Klik op de eerste link. De debugger stopt deze keer niet aan het breakpoint. *OnInitialized()* wordt dus niet uitgevoerd.
- Klik op de tweede link. *OnInitialized()* wordt opnieuw niet uitgevoerd.
- Tik in de browser de URL */dobbelen/2* en druk op Enter. De debugger stopt opnieuw tot tweemaal toe aan de breakpoint.

Waarom wordt de method *OnInitialized()* twee keer uitgevoerd? In een Blazor app wordt de app geprerenderd op de server en daarna nog eens gerenderd op de client. De prerendering moet je zien als een soort voorbereiding op de rendering op de client. Bij elke rendering wordt de method OnInitialized() uitgevoerd.

OnParametersSet/OnParametersSetAsync

De method *OnParametersSet* of *OnParametersSetAsync* wordt uitgevoerd wanneer de waarde van één of meerdere parameters verandert via zijn parent. Deze method wordt dus mogelijk meerdere keren uitgevoerd. Blazor herberekent dan de zogenaamde rendertree en voert daarna *OnAfterRender()* uit.

Rendertree

Het weergeven van een responsive layout zorgt voor heel veel aanpassingen aan de DOM. Dit werkt zeer vertragend. Sommige clientside tools werken daarom met een Virtual Dom en een Incremental DOM.

Een virtual DOM is een in-memory representatie van de elementen waarmee de HTML-pagina is opgebouwd. Deze tree van elementen wordt opgebouwd door de blazor componenten via de virtual method *BuildRenderTree*. Het voordeel is dat de ganse opbouw in-memory gebeurt en de browser enkel het eindresultaat hoeft weer te geven.

Na een update van de pagina kan de nieuwe rendertree vergeleken worden met de oude en moet enkel het verschil geüpdatet worden. Dit verschil wordt ook wel de *diff* genoemd. Bij een Blazor Server project moet dan enkel deze diff doorgestuurd worden naar de client wat tijdswinst betekent.

We proberen dit uit.

- Override in de component *Child.razor* de method *OnParametersSet()* door onderstaande code toe te voegen :

```
protected override void OnParametersSet()
{
    base.OnParametersSet();
}
```

- Zet een breakpoint in deze method.
- Start de app met debugging en kies in het menu voor *Data binding* of browse naar */parent*. De app stopt aan het breakpoint. *OnParametersSet()* wordt uitgevoerd want de parameter in *Child* krijgt zijn initiële waarde (0) vanuit de parent.
- Klik op *Continue* om verder te gaan. De uitvoering van de app stopt opnieuw bij het breakpoint. De method *OnParametersSet()* wordt dus eveneens twee maal uitgevoerd.
- Ga verder met de uitvoering en klik op de knop *Verhoog met 5. OnParametersSet()* wordt opnieuw uitgevoerd want de parameter in *Child* wordt aangepast vanuit de parent.

OnAfterRender/OnAfterRenderAsync

De method *OnAfterRender* of *OnAfterRenderAsync* wordt opgeroepen nadat de parentcomponent de waarde van één of meerdere parameters heeft veranderd en tengevolge daarvan de rendertree is veranderd. Deze rendertree kan echter ook gewijzigd worden na het klikken op een button in de component zelf of nadat de component een *StateHasChanged()*-method heeft opgeroepen (zie verder).

De method *OnAfterRender(Async)* heeft een parameter *firstRender*. De waarde is *true* bij een eerste keer dat de component wordt weergegeven, daarna *false*. Op basis van deze waarde kan je beslissen om al dan niet een éénmalige actie uit te voeren.

Verderop in deze cursus staan enkele voorbeelden waarin we de method *OnAfterRender* overriden en de parameter *firstRender* gebruiken.

StateHasChanged

Na een klik op een button wordt de *StateHasChanged*-method automatisch opgeroepen. Dit zorgt voor een rerender van de page en ook voor de uitvoering van de *OnAfterRender()*-method. In sommige gevallen moet de page ook gerenderd worden maar wordt de *StateHasChanged*-method niet automatisch uitgevoerd. Denk maar aan een nodige wijziging in de pagina na een bepaalde tijdspanne. Je roept dan zelf de method *StateHasChanged* op. De page wordt dan gerenderd.

ShouldRender

De method *ShouldRender()* retourneert een boolean. Is het resultaat *false* dan wordt de page na een eerste rendering niet meer aangepast.

Een voorbeeld :

- Voeg aan de component *Eenentwintig* onderstaande code toe :

```
protected override bool ShouldRender()
{
    return currentScore <= 21;
}
```

Deze method geeft aan dat de page moet gerenderd worden zolang de score kleiner is of gelijk aan 21. Wordt de score hoger dan zal het klikken op de knop *Trek een kaart* geen zichtbaar gevolg hebben. Dit betekent niet dat de score niet verder oploopt !

- Zet in de codesectie (of de aparte baseclass) de lijn code die een exception throwt bij een score groter dan 21 in commentaar.
- Probeer uit. Vanaf 22 of hoger worden geen extra kaarten getoond.

IDisposable

Je kan een component de interface *IDisposable* laten implementeren. De method *Dispose()* wordt dan opgeroepen van zodra de component uit de rendertree is verdwenen.

Verderop in deze cursus vind je enkele voorbeelden.

7 Local Storage en Session Storage

In ASP.NET Core kan je gegevens over meerdere pagina's heen onthouden door ze op te slaan in session state. Een Blazor app werkt echter helemaal anders en ondersteunt geen Session State.

Een goed alternatief is Session Storage en Local Storage. Deze heb je misschien reeds leren kennen in de JavaScript cursus.

7.1 Session Storage

Als voorbeeld van Session Storage gaan we het getal in de Counter page onthouden door het op te slaan in Session Storage. Op deze manier wordt de waarde van de counter niet telkens op 0 gereset wanneer we binnen dezelfde sessie terugkeren naar deze page.

Om te werken met Local Storage in Blazor gebruiken we de NuGet package *Blazored.SessionStorage* van Chris Sainty. Je installeert deze package in het project waarin de component staat die de session storage gebruikt.

- Installeer de NuGet package *Blazored.SessionStorage*. Als jouw Counter-component in een .Client-project staat dan installeer je de NuGet package in het .Client-project.

We voegen in *Program.cs* de service *BlazoredSessionStorage* toe. Wanneer jouw app een apart .Client-project heeft dan voeg je de service toe in de Program.cs van het .Client-project.

- Bij Blazor apps zonder .Client-project wijzig je de code in *Program.cs* als volgt :

```
using Blazored.SessionStorage;
…
builder.Services.AddBlazoredSessionStorage();
var app = builder.Build();
…
```

- In Blazor apps mét .Client-project wijzig je de code in *Program.cs* als volgt :

```
using Blazored.SessionStorage;
…
builder.Services.AddBlazoredSessionStorage();
await builder.Build().RunAsync();
```

We kunnen de service nu injecteren in de component *Counter.*

- Voeg in *Counter.razor* onder de @page-directive onderstaande @inject-directive toe :

```
@page "/counter"
@inject Blazored.SessionStorage.ISessionStorageService sessionStore
<PageTitle>Counter</PageTitle>
…
```

We moeten nu twee aanpassingen doorvoeren in de code : in de method *IncrementCount()* bewaren we de waarde van *currentCount* in Session Storage nadat de waarde werd verhoogd met 1. Als tweede aanpassing moet er bij het laden van de component gekeken worden in Session Storage of er een waarde werd opgeslagen. Als dat zo is dan moet de counter op deze waarde geïnitialiseerd worden. Zoniet dan wordt de initiële waarde 0.

We beginnen met het bewaren van de counterwaarde nadat deze werd verhoogd.

- Wijzig de method *IncrementCount()* als volgt :

```
private async Task IncrementCountAsync()
{
    currentCount++;
    await sessionStore.SetItemAsync<int>("counter", currentCount);
}
```

We bewaren de waarde van *currentCount* in Session Storage met de method *SetItemAsync<T>* van de geïnjecteerde sessionstorageservice. Hierbij is *T* het type van de te bewaren variabele. Als parameters geven we de naam voor de Session Storage variabele mee en ook de waarde.

Aangezien we een async-method *SetItemAsync<T>* gebruiken maken we ook van de method *IncrementCount* een async-method. We geven dit aan met het keyword *async*. De method retourneert nu een Task i.p.v. een void. Het is ook een goede gewoonte async-methods een naam te geven die eindigt op -Async. Je hernoemt de method dus in *IncrementCountAsync.*

Ook de methodoproep moet asynchroon gebeuren.

- Wijzig in de layout de code voor de button als volgt :

```
<button class="btn btn-primary" @onclick="async () => await
    IncrementCountAsync()">Click me</button>
```

Opmerking

In een Blazor WebAssembly app kan je ook op **synchrone** wijze gegevens bewaren (en ophalen) in Session Storage. Je injecteert bovenaan de code een *ISyncSessionStorageService.* Je bewaart dan een gegeven op Session Storage met de method *SetItem<T>*, ophalen doe je met *GetItem<T>.*

Wanneer de component wordt geladen halen we de opgeslagen waarde van de counter op uit de Session Storage. Een Blazor app wordt geprerenderd en dus kunnen we pas JavaScript-code uitvoeren na een eerste rendering. Het ophalen van de waarde zal dus moeten gebeuren in de method *OnAfterRenderAsync()* en niet in *OnInitializedAsync().*

- Voeg onderstaande code toe in de code-section :

```
protected override async Task OnAfterRenderAsync(bool firstRender)
{
    if (firstRender)
    {
        int? sessionCounter = await sessionStore.GetItemAsync<int>("counter");
        if (sessionCounter == null)
            currentCount = 0;
        else
            currentCount = (int)sessionCounter;
        StateHasChanged();
    }
}
```

Wanneer de component een eerste keer wordt gerenderd (dus niet na een aanpassing van de teller) halen we een int-waarde met de naam *counter* op uit Session Storage. Omdat het mogelijk is dat deze niet wordt gevonden slaan we het resultaat op in een nullable-int.

Als de waarde niet werd gevonden dan initialiseren we de counter op 0. Vinden we wél een waarde in Session Storage dan converteren we die naar een int en slaan die op in de variabele currentCount.

De nieuwe waarde van currentCount wordt pas weergegeven op het scherm na een oproep van de method *StateHasChanged().*

- Probeer uit : start de app en kies in het menu voor *Counter.* De waarde is 0.
- Klik enkele keren op de knop *Click me.* De waarde verhoogt.
- Kies een ander menuitem en keer daarna terug naar *Counter.* De waarde werd bewaard.

7.2 Local Storage

Je kan een gegeven ook bewaren in Local Storage. De waarde wordt dan op de harde schijf bewaard en niet in het RAM-geheugen van de client. Op deze manier blijft de waarde ook bewaard voor een volgende session.

Deze keer laten we de gebruiker zijn naam invullen op de Home-page en bewaren. Van zodra de naam gekend is wordt de naam weergegeven en kan deze terug gewist worden.

We beginnen opnieuw met het toevoegen van een NuGet package. Voor Local Storage installeren we de NuGet package *Blazored.LocalStorage.*

- Installeer de NuGet package *Blazored.LocalStorage*.
- Bij alle Blazor apps (mét of zonder .Client-project) wijzig je de code in *Program.cs* in het project zonder .Client-extensie als volgt :

```
using Blazored.LocalStorage;
…
builder.Services.AddBlazoredLocalStorage();
var app = builder.Build();
…
```

- In Blazor apps mét .Client-project wijzig je bovendien de code in *Program.cs* in het .Client-project als volgt :

```
using Blazored.LocalStorage;
…
builder.Services.AddBlazoredLocalStorage();
await builder.Build().RunAsync();
```

We passen nu de pagelayout aan.

- Vervang de layout-code in *Home.razor* door onderstaande code :

```
@page "/"
@inject Blazored.LocalStorage.ILocalStorageService localStore

@if (!naamgekend) {
    <label for="naam" class="form-label">Jouw naam</label>
    <input id="naam" type="text" @bind="naam" class="form-control" />
    <button class="btn btn-success mt-3"
        @onclick="async () => await SaveLocalAsync()">Naam opslaan</button>
}
else {
    <span>Hallo @naam</span>
    <button class="btn btn-danger ms-3"
        @onclick="async () => await ClearLocalAsync()">Naam wissen</button>
}
```

In bovenstaande code injecteren we een ILocalStorageService en noemen die localStore. Vervolgens tonen we ofwel een inputveld voor de naam en een Save-knop, ofwel de bewaarde naam en een Wissen-knop. We laten dit afhangen van de waarde van een boolean-variabele *naamgekend.*

- Voeg onderstaande code-section toe :

```
@code {
    private string? naam;
    private bool naamgekend = false;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            naam = await localStore.GetItemAsync<string>("usernaam");
            if (naam != null) {
                naamgekend = true;
                StateHasChanged();
            }
        }
    }

    public async Task SaveLocalAsync()
    {
        await localStore.SetItemAsync<string>("usernaam", naam);
        naamgekend = true;
    }

    public async Task ClearLocalAsync()
    {
        naam = null;
        naamgekend = false;
        await localStore.RemoveItemAsync("usernaam");
    }
}
```

We definiëren twee variabelen : een string *naam* die gekoppeld is met het inputveld en een boolean *naamgekend* die aanvankelijk op false staat. Op die manier worden het inputveld en de Save-knop getoond.

Bij een eerste weergave van de component wordt een item *usernaam* opgezocht in Local Storage. Als dat iets oplevert dan wordt de waarde van de boolean *naamgekend* true. Na een oproep van de method *StateHasChanged()* wordt de naam getoond, samen met de Wis-knop.

Wanneer er op de bewaarknop wordt geklikt slaan we de ingevulde naam op in Local Storage onder de naam *usernaam.* De boolean wordt gewijzigd.

Om de naam te wissen gebruiken we de method *RemoveItemAsync().* Als parameter geven we de naam van het te wissen item mee. Je kan ook drastischer tewerk gaan en de Local Storage volledig leeg maken met de method *ClearAsync().*

- Je kan opnieuw uitproberen : start de app.
- Vul jouw naam in en sla op. Je krijgt meteen jouw naam te zien en ook de wis-knop.
- Bezoek enkele andere menuitems en keer terug naar de Home-page. Je ziet nog steeds jouw naam.
- Stop de app en start opnieuw op. Je ziet opnieuw jouw naam.

7.3 Objecten wegschrijven

We kunnen ook objecten of verzamelingen van objecten wegschrijven naar Local en Session Storage. We doen dit door het object of de verzameling te serializeren naar een JSON-string vooraleer we het wegschrijven in Storage. Bij het ophalen van de gegevens moeten we de JSON-string terug deserializen.

Als voorbeeld maken we een component waarmee we een todo-list bijhouden.

- Voeg in de folder Pages een nieuwe razor component toe en noem deze *TodoList.razor.*
- Breid het menu in *NavMenu.razor* uit met een extra item :

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="todo">
        <span class="bi bi-clipboard-check" aria-hidden="true"></span> Todolist
    </NavLink>
</div>
```

- Voeg onderstaande code toe aan *TodoList.razor* :

```
@page "/todo"
@inject Blazored.LocalStorage.ILocalStorageService localStore
@using System.Text.Json

<h3>TodoList</h3>
<label for="todo" class="form-label">Nieuwe taak</label>
<input id="todo" @bind="nieuweTodo" type="text" class="form-control"/>
<button class="btn btn-primary my-3"
    @onclick="async () => await SaveTodoAsync()">Save</button>

@foreach (var taak in todos) {
    <p><img src="images/delete.png" class="me-2"
        @onclick="async () => await DeleteTodoAsync(taak)"/>@taak</p>
}

@code {
    private string? nieuweTodo;
    private List<string> todos = new List<string>();

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender) {
            var geserialiseerdelijst =
                await localStore.GetItemAsync<string>("todos");
            if (geserialiseerdelijst != null)
                todos = JsonSerializer.Deserialize<List<string>>(
                    geserialiseerdelijst);
            StateHasChanged();
        }
    }

    public async Task SaveTodoAsync()
    {
        if (nieuweTodo != string.Empty) {
            todos.Add(nieuweTodo);
            nieuweTodo = string.Empty;
            await localStore.SetItemAsync<string>("todos",
                JsonSerializer.Serialize(todos));
        }
    }
```

```
        public async Task DeleteTodoAsync(string taak)
        {
            todos.Remove(taak);
            await localStore.SetItemAsync<string>("todos",
                JsonSerializer.Serialize(todos));
        }
    }
```

We injecteren terug een ILocalStorageService en voegen ook een using-statement toe voor de namespace *Newtonsoft.Json.*

We databinden een inputveld met een stringvariabele *nieuweTodo.* Daarnaast komt een knop waarmee we de nieuwe taak toevoegen en de todo-lijst opslaan in local storage.

Daaronder wordt de todo-lijst weergegeven. Vóór elke taak wordt een kruisje afgebeeld. Een klik hierop verwijdert de taak en bewaart de aangepaste lijst.

In de code-section bewaren we de taken in een List<string>.

Bij een eerste weergave halen we de takenlijst op uit storage. Als die gevonden wordt dan deserializen we de JSON-string naar een List<string> en stoppen we het resultaat in de variabele *todos.* Om de UI bij te werken roepen we de method *StateHasChanged* op.

Voor het bewaren van een nieuwe taak schrijven we een method *SaveTodoAsync()*. Daarin controleren we of er iets werd ingetikt in het inputveld. Als dat zo is dan voegen we de taak toe aan de lijst en wissen we het inputveld. Tenslotte bewaren we de nieuwe lijst door deze eerst te serializen naar een string en dan op te slaan in storage.

De method *DeleteTodoAsync()* krijgt de te verwijderen taak mee als parameter. We verwijderen de taak uit de List en bewaren de List vervolgens in storage.

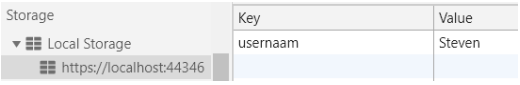
- Voeg in wwwroot in de folder *images* de afbeelding *delete.png* toe.
- Start de applicatie en probeer enkele taken toe te voegen. Verwijder er eventueel ook één.
- Sluit de applicatie en start deze opnieuw op. Je ziet terug de lijst.

7.4 Protected browser storage

Local Storage heeft één groot nadeel : de gebruiker kan het opgeslagen gegeven heel eenvoudig wijzigen. We proberen dit even uit.

- Start de app. Vul op de Home-page jouw naam in en klik op de knop *Naam opslaan.*
- Druk in de browser op F12 om de Developer Tools te zien.

Wanneer je in het Developer Tools venster op het tabblad *Application* onder *Storage – Local Storage* klikt op de URL van de app zie je de opgeslagen waarde voor *usernaam* :



- Wijzig nu in het Developer Tools venster de Value voor usernaam in Prosper.
- Refresh de webpagina. De webpagina toont nu de nieuwe naam.

Deze vrijheid willen we de gebruiker niet altijd geven. We kunnen dit sinds .NET 5 oplossen met zogenaamde protected browser storage. Met deze techniek, die voorlopig enkel toepasbaar is in een Blazor Server App, wordt de data op een geëncrypteerde manier bewaard in het geheugen (sessionstorage) of op de schijf (localstorage).

We passen onze applicatie aan.

- Wijzig de code in Home.razor als volgt :

```
@page "/"
@using Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage
@inject ProtectedLocalStorage localStore
…
@code {
    private string? naam;
    private bool naamgekend = false;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender) {
            try {
                var result = await localStore.GetAsync<string>("usernaam");
                if (result.Success) {
                    naam = result.Value;
                    naamgekend = true;
                    StateHasChanged();
                }
            }
            catch (Exception ex) {
                naam = null;
                naamgekend = false;
                StateHasChanged();
            }
        }
    }

    public async Task SaveLocalAsync()
    {
        await localStore.SetAsync("usernaam", naam);
        naamgekend = true;
    }

    public async Task ClearLocalAsync()
    {
        naam = null;
        naamgekend = false;
        await localStore.DeleteAsync("usernaam");
    }
}
```

In de layout veranderen we dus niks. We injecteren enkel een ProtectedLocalStorage service en noemen deze *localStore*. Daarvoor moeten we wel een using statement toevoegen.

In de method OnAfterRenderAsync() stoppen we de code die de usernaam probeert op te halen in een try-block. Het is namelijk zo dat wanneer de gebruiker de opgeslagen waarde wijzigt via de Developer Tools en deze niet geëncrypteerde waarde vervolgens wordt ingelezen, er een exception wordt geworpen.

Om de waarde op te halen gebruiken we de method GetAsync<>(). In dit geval halen we een string op dus gebruiken we de method GetAsync<string>(). Dit levert ons een resultaat op in de vorm van een ProtectedBrowserStorageResult (en niet direct de stringwaarde). Deze heeft een Success-property. Is die true dan halen we de stringwaarde uit de Value-property en passen we de UI aan.

Liep er iets mis met het ophalen van de naam dan maken we de naam leeg en passen we de UI aan.

In de method SaveLocalAsync() bewaren we de naam door de method SetAsync() te gebruiken. Om de naam te wissen uit localstorage gebruiken we de method DeleteAsync().

We proberen dit uit.

- Start de app en druk F12 voor de Developer Tools. Je ziet dat er nog steeds een naam is bewaard op local storage. Deze naam is niet geëncrypteerd en wordt dan ook niet weergegeven op de homepage.
- Tik een naam in het inputveld en klik op Naam opslaan. De naam wordt op de page weergegeven en in de Developer Tools zie je een geëncrypteerde value bij de key *usernaam*.
- Wijzig de naam in de Developer Tools en refresh de homepage. De naam is geen geldig geëncrypteerde waarde en dus wordt er een exception geworpen. Die vangen we op, de code in het catch-block wordt uitgevoerd.

7.5 Oefening

Pas de counter-component aan zodat de currentcount met protected browser storage wordt opgeslagen in localstorage.

8 CSS isolation

Op dit moment wordt onze applicatie opgemaakt op basis van één of meerdere css-bestanden waarnaar in App.razor wordt gerefereerd.

Sinds de introductie van .NET 5 is het mogelijk om een component te voorzien van een eigen css-bestand. Dat is bijvoorbeeld bij de layout componenten MainLayout en NavMenu reeds het geval.

Door een component te voorzien van een eigen css-bestand kan je deze component een afwijkende opmaak geven t.o.v. de standaardopmaak die via *app.css* is toegekend.

8.1 Een eenvoudig voorbeeld

We proberen dit uit in een eenvoudig voorbeeld. We geven een h1-tag een groene tekstkleur via het css-bestand *app.css*. Aan de component *eenentwintig* voegen we een .razor.css-bestand toe waarin we afwijkende opmaak noteren.

- Voeg in het css-bestand *app.css* uit de folder *wwwroot* onderstaande css-rule toe :

```
h1 {
    color:green;
}
```

- Voeg in de folder *Pages* een bestand toe met de naam *Eenentwintig.razor.css* en noteer er onderstaande code in :

```
h1 {
    color:red;
}
```

- Start de app. In alle componenten worden de <h1>-tags in het groen weergegeven behalve in de component *eenentwintig.razor,* daar is de <h1>-tag rood. Als jouw h1-header niet gekleurd worden dan zal je wellicht de cache even moeten ledigen.

8.2 Het ::deep pseudo-element

De css die we noteren in het aparte .razor.css-bestand geldt enkel en alleen voor de gelijknamige component waar het css-bestand bij hoort. Eventuele childcomponenten erven deze css niet.

Wil je toch dat childcomponenten de css erven dan moet je twee speciale maatregelen nemen :

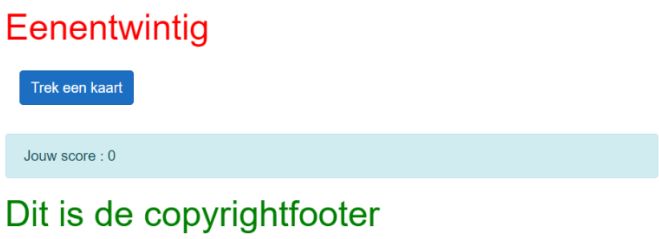
1. Je voegt in het .razor.css-bestand van de parent een extra css-rule toe met een ::deep pseudo-element.
2. Je wrapt het child-element in een .

We proberen dit uit. De component *Eenentwintig.razor* bevat een childelement *CopyrightFooter*. In die component voegen we een h1-element toe. We bekijken eerst hoe het h1-element wordt weergegeven in het childelement zonder extra maatregelen. Daarna voegen we de nodige wijzigingen toe zodat de opmaak van *Eenentwintig* wordt overgeërfd in *CopyrightFooter*.

- Voeg helemaal bovenaan in *CopyrightFooter.razor* een h1-element toe :

```
<h1>Dit is de copyrightfooter</h1>
```

- Probeer uit. De childcomponent heeft de defaultopmaak uit *site.css* en niet deze van de parent.



We brengen nu de nodige aanpassingen aan in *Eenentwintig.razor.css* zodat de component *CopyrightFooter* de css erft.

- Voeg in *Eenentwintig.razor.css* onderstaande extra style rule toe :

```
::deep h1 {
    color: red;
}
```

- Zet in *Eenentwintig.razor* de component *CopyrightFooter* binnen een -element :

```
<div>
    <CopyrightFooter @attributes="@footerAttributes" @ref="cfdynamic">
        Blazor rocks
    </CopyrightFooter>
</div>
```

- Probeer uit. De childcomponent heeft niet langer de default opmaak maar de afwijkende opmaak gedefinieerd in de parent met ::deep.

We hebben nu in *Eenentwintig.razor.css* de stylerule voor het childcomponent herhaald met een pseudo-element ::deep. De stylerule hebben we volledig overgenomen. Maar wat als we daar nóg een andere waarde in noteren? We proberen het uit.

- Wijzig in *Eenentwintig.razor.css* de extra style rule als volgt :

```
::deep h1 {
    color: blue;
}
```

- Probeer uit. De childcomponent heeft de afwijkende opmaak die in de parent gedefinieerd werd met het pseudo-element ::deep.

9 Layouts

In een webapplicatie komen onderdelen zoals een menu, een footer of titel vaak op alle pages terug. Om te vermijden dat we deze stukken code op elke page opnieuw moeten schrijven kunnen we ze beter opnemen in een layout. Dit kan ook in Blazor.

In Blazor is een layout een component die erft van LayoutComponentBase waardoor het beschikt over een property *Body*. Deze property bevat de (variabele) content die in de layout wordt weergegeven.

9.1 Default layout

Je kan de default layout instellen in *Routes.razor* :

```
<Router AppAssembly="typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="routeData" DefaultLayout="typeof(Layout.MainLayout)" />
        <FocusOnNavigate RouteData="routeData" Selector="h1" />
    </Found>
</Router>
```

In bovenstaande code wordt een default layout ingesteld voor de url's waarvoor een route wordt gevonden.

9.2 Afwijken van de default layout

We kunnen voor één specifieke component of voor alle componenten uit een bepaalde folder afwijken van de default layout.

Wil je voor een bepaalde folder een afwijkende layout dan voeg je in die folder een component met de naam *_Imports.razor* toe. In die component, waarin je uiteraard ook extra using-statements kan plaatsen, vermeld je de gewenste layout met een @layout directive.

Wil je voor één bepaalde component de layout instellen dan gebruik je de directive @layout in de component zelf. Voorbeeld :

```
@layout MijnLayout
@page "/"
<h1>Hello World!</h1>
…
```

We testen dit uit.

- Maak in de folder *Pages* een nieuwe folder *MetAndereLayout.*
- Voeg in deze folder twee nieuwe componenten toe en noem ze *Component1.razor* en *Component2.razor.*
- Voeg in *Component1.razor* bovenaan een route toe :

  ```
  @page "/component1"
  <h3>Component1</h3>
  …
  ```

- Doe hetzelfde in *Component2.razor* :

  ```
  @page "/component2"
  <h3>Component2</h3>
  …
  ```

Voor deze twee nieuwe componenten zullen we een andere layout gebruiken dan de layout *MainLayout.razor.*

We maken eerst een extra layout aan en voegen dan de component *_Imports.razor* toe in de folder *MetAndereLayout.*

- Voeg in de folder *Layout* een razor component toe en noem deze *AndereLayout.razor.*
- Vervang de code in *AndereLayout.razor* door de onderstaande code :

```
@inherits LayoutComponentBase
<PageTitle>Blazor App</PageTitle>
<div class="page">
    <div class="sidebar">
        <NavMenu />
    </div>
    <main>
        <article class="content px-4">
            @Body
        </article>
    </main>
</div>
```

Bovenstaande layout is vrijwel identiek aan *MainLayout* maar zonder de toprow met daarin de About-link.

De default layout heeft ook een bijhorende css-file. We kopiëren nu de inhoud ervan naar een nieuwe css-file voor de layout AndereLayout.razor.

- Voeg in de folder *Layout* een nieuw css-bestand toe en noem het *AndereLayout.razor.css*.
- Kopieer de volledige inhoud van *MainLayout.razor.css* en plak het in *AndereLayout.razor.css*.
- Breng een kleine wijziging aan in *AndereLayout.razor.css.* Voeg een achtergrondkleur toe voor het element .page :

```
.page {
    position: relative;
    display: flex;
    flex-direction: column;
    background-color: lightgray;
}
```

We activeren de layout AndereLayout nu voor de componenten in de folder MetAndereLayout.

- Voeg in de folder *MetAndereLayout* een component *_Imports.razor* toe.
- Vervang de inhoud van deze component door onderstaande code :

```
@using BlazorServer.Components.Layout
@layout AndereLayout
```

In het bovenstaande using-statement vervang je BlazorServer door de naam van jouw Blazor-project.

Door deze code op te nemen in een *_Imports.razor* bestand geven we aan dat de layout *AndereLayout* moet gebruikt worden voor alle componenten in de folder *MetAndereLayout*.

- Start de app en browse naar */component1*. De component wordt opgemaakt met de andere layout.
- Browse naar */component2*. Ook deze component wordt met de afwijkende layout opgemaakt.

We laten component2 nu opnieuw weergeven met de layout *MainLayout* door deze layout expliciet aan te geven in de component zelf.

- Wijzig de code in *Component2.razor* als volgt :

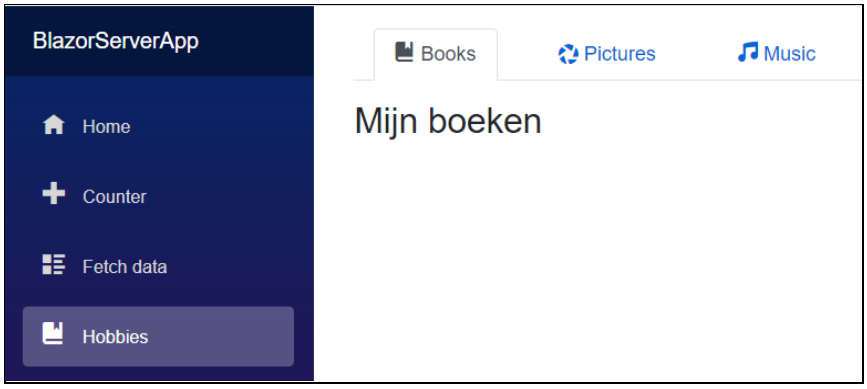
```
@page "/component2"
@layout MainLayout
…
```

- Start de app opnieuw en browse naar *Component1* en naar *Component2.* Component1 wordt nog steeds weergegeven met de afwijkende layout, Component2 met de 'normale' MainLayout.

9.3 Geneste layouts

Je kan layouts ook nesten. Dit kan handig zijn als jouw app naast een hoofdnavigatiemenu bijvoorbeeld ook een subnavigatiemenu heeft. We proberen dit uit.

We beogen het onderstaande resultaat. Wanneer we in het hoofdmenu klikken op *Hobbies* dan zien we de eerste hobbypagina : *Books.* Via een horizontaal submenu kunnen we de andere hobby-pagina's bereiken: *Pictures* en *Music*.



- Maak in de folder *Pages* drie nieuwe componenten aan : *Books.razor, Pictures.razor* en *Music.razor.*
- Neem onderstaande code over in *Books.razor* :

```
@layout SubLayout
@page "/books"
<h3>Mijn boeken</h3>
```

- Neem onderstaande code over in *Pictures.razor* :

```
@layout SubLayout
@page "/pictures"
<h3>Mijn foto's</h3>
```

- Neem onderstaande code over in *Music.razor* :

```
@layout SubLayout
@page "/music"
<h3>Mijn muziek</h3>
```

Met de directive @layout geven we dus aan dat de layout *SubLayout* moet gebruikt worden. De inhoud van de bovenstaande componenten zal telkens in *SubLayout* worden ingevoegd op de plaats waar *@Body* staat.

- Maak in de folder *Layout* een nieuwe razor component en noem die *SubLayout.razor.*
- Voorzie deze component van onderstaande code :

```
@layout MainLayout
@inherits LayoutComponentBase
<ul class="nav nav-tabs mb-3">
    <li class="nav-item px-3">
        <NavLink class="nav-link" href="/books">
            <span class="bi bi-book" aria-hidden="true"></span> Books
        </NavLink>
    </li>
    <li class="nav-item px-3">
        <NavLink class="nav-link" href="/pictures">
            <span class="bi bi-images" aria-hidden="true"></span> Pictures
        </NavLink>
    </li>
    <li class="nav-item px-3">
        <NavLink class="nav-link" href="/music">
            <span class="bi bi-music-note-beamed" aria-hidden="true"></span>
Music
        </NavLink>
    </li>
</ul>
@Body
```

Met de code `@layout MainLayout` geven we aan dat deze component moet weergegeven worden met de MainLayout. Dit betekent dat alle content van deze component in MainLayout moet geplaatst worden op de plaats waar *@Body* staat.

De component *SubLayout* is een layout dus laten we de component erven van *LayoutComponent-Base*. Daaronder geven we een horizontaal menu weer door gebruik te maken van de bootstrap class nav-tabs.

- Voeg in _Imports.razor een extra using-statement toe zodat de SubLayout wordt gevonden :

```
@using BlazorServer.Components.Layout
```

In het hoofdmenu voegen we nu een menuitem toe dat naar de eerste hobby-pagina verwijst : books.

- Voeg in *NavMenu.razor* onderstaand listitem toe in de lijst :

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="books">
        <span class="oi oi-book" aria-hidden="true"></span> Hobbies
    </NavLink>
</div>
```

- Probeer uit.

10 Forms en validation

In dit hoofdstuk bekijken we hoe we in Blazor forms opbouwen en valideren.

10.1 EditForm

In Blazor kan je op de vertrouwde manier een form opbouwen met een <form>-tag en inputvelden. Je kan echter ook gebruik maken van de Blazor-component *EditForm*. Deze beschikt over tal van ingebouwde voorzieningen met betrekking tot validatie van de form.

Als voorbeeld maken we een formulier waarmee een gebruiker zich registreert door zijn voornaam en familienaam in te vullen.

Het is een goede gewoonte een viewmodel-class te maken voor de formuliergegevens.

- Maak in een nieuwe folder *ViewModels* een class *UserViewModel.cs.*
- Voeg onderstaande code toe :

```
using System.ComponentModel.DataAnnotations;
…
public class UserViewModel
{
    [Required(ErrorMessage = "{0} is een verplicht veld")]
    [StringLength(30, ErrorMessage =
        "Gebruik maximum {1} tekens voor het veld {0}")]
    public string Voornaam { get; set; }

    [Required(ErrorMessage = "{0} is een verplicht veld")]
    [StringLength(30, ErrorMessage =
        "Gebruik maximum {1} tekens voor het veld {0}")]
    public string Familienaam { get; set; }
}
```

We gebruiken annotations om de nodige validatieregels aan te geven.

We ontwerpen nu het formulier.

- Maak in de folder *Pages* een nieuwe component en noem deze *Register.razor.*
- Voeg in *NavMenu* een listitem toe voor deze component :

```
…
<div class="nav-item px-3">
    <NavLink class="nav-link" href="register">
        <span class="bi bi-pencil" aria-hidden="true">
        </span> Registratieformulier
    </NavLink>
</div>
…
```

- Voorzie de component *Register* van onderstaande code :

```
@page "/register"
<h1>Registratieformulier</h1>

<EditForm Model="@userViewModel" OnValidSubmit="HandleValidSubmit">
    <DataAnnotationsValidator />
```

```
        <div class="row my-3">
            <div class="col-3">
                <label for="voornaam" class="col-form-label">
                    Voornaam
                </label>
                <InputText id="voornaam" @bind-Value="userViewModel.Voornaam"
                           class="form-control" />
            </div>
            <div class="col-3">
                <label for="familienaam" class="col-form-label">
                    Familienaam
                </label>
                <InputText id="familienaam" @bind-Value="userViewModel.Familienaam"
                           class="form-control" />
            </div>
        </div>
        <ValidationSummary />
        <button type="submit" class="btn btn-primary">Submit</button>
    </EditForm>

    @code {
        private UserViewModel userViewModel = new UserViewModel();
        private void HandleValidSubmit()
        {
            //formulier wordt verwerkt
        }
    }
```

- Voeg in _Imports.razor een extra using-statement toe naar de folder ViewModels zodat in de bovenstaande code-sectie de class UserViewModel wordt herkend.

Een woordje uitleg bij deze code. In de code-section maken we een viewmodel instance *userViewModel* aan en bewaren die in een private membervariabele. De class UserViewModel is het model voor de form. We geven dit in de form aan via het EditForm-attribuut ***Model***.

Het formulier wordt afgehandeld met de method *HandleValidSubmit().* Dit kan in de EditForm-tag op 2 manieren : ofwel met de eventcallback ***OnValidSubmit,*** ofwel via *OnSubmit.* Het verschil is dat als je *OnValidSubmit* gebruikt je eventueel ook nog supplementair met *OnInvalidSubmit* een method kan opgeven die moet uitgevoerd worden wanneer het formulier niet geldig valideert.

Met de component ***DataAnnotationsValidator*** geef je aan dat het formulier de data-annotations uit de modelclass moet gebruiken voor de validatie. Zonder deze component heb je geen validatie. Vergeet deze dus niet toe te voegen !

De component ***ValidationSummary*** lijst alle validationerrormessages op.

Je kan de errormessages ook individueel bij de foutief validerende inputvelden plaatsen door gebruik te maken van een ***ValidationMessage***-component :

```
<ValidationMessage For="@(()=> userViewModel.Familienaam)" />
```

Voor het inputveld gebruiken we geen input-tag maar een ***InputText***-component. Je gebruikt two-way data binding tussen de InputText-component en de bijhorende property van het model door @bind-Value toe te voegen.

Door de component *EditForm* te gebruiken krijgen inputvelden die geldig valideren automatisch een css-class *valid.* Velden die niet geldig valideren krijgen de css-class *invalid.*

Een mogelijk resultaat :

10.2 Inputcomponenten

In het bovenstaande voorbeeld gebruikten we een **InputText**-component om de voornaam en familienaam te laten ingeven. Je gebruikt een InputText-component voor een <input>-tag met het type *text*. Voor inputvelden met een ander type kan je een component gebruiken uit onderstaande lijst. In de rechtse kolom zie je welke html-code er voor de inputcomponent wordt gerenderd.

Input component	**wordt gerenderd als…**
`InputTextArea`	`<textarea>`
`InputSelect`	`<select>`
`InputNumber`	`<input type="number">`
`InputCheckbox`	`<input type="checkbox">`
`InputDate`	`<input type="date">`
`InputFile`	`<input type="file">`

Aan deze inputcomponenten kan je de gebruikelijke attributen toevoegen die je anders ook voor een input-, textarea- of select-tag zou gebruiken.

We vullen ons voorbeeld aan met drie extra velden.

- Voeg in de class *UserViewModel* de property's *Geboortedatum, Opleidingscentrum* en *Rijbewijs* toe :

```
public class UserViewModel
{
    [Required(ErrorMessage = "{0} is een verplicht veld")]
    [StringLength(30, ErrorMessage =
        "Gebruik maximum {1} tekens voor het veld {0}")]
    public string Voornaam { get; set; }
    [Required(ErrorMessage = "{0} is een verplicht veld")]
    [StringLength(30, ErrorMessage =
        "Gebruik maximum {1} tekens voor het veld {0}")]
    public string Familienaam { get; set; }
    public DateTime GeboorteDatum { get; set; }
    public string Opleidingscentrum { get; set; }
    public bool Rijbewijs{ get; set; }
}
```

- Pas het formulier als volgt aan :

```
@page "/register"
<h1>Registratieformulier</h1>
<EditForm Model="@userViewModel" OnValidSubmit="HandleValidSubmit">
    <DataAnnotationsValidator />
    <div class="row my-3">
        <div class="col-3">
            <label for="voornaam" class="col-form-label">Voornaam</label>
            <InputText id="voornaam" @bind-Value="userViewModel.Voornaam"
                       class="form-control" />
        </div>
        <div class="col-3">
            <label for="familienaam" class="col-form-label">
                Familienaam
            </label>
            <InputText id="familienaam" @bind-Value="userViewModel.Familienaam"
                       class="form-control" />
        </div>
    </div>
    <div class="row mb-4">
        <div class="col-2">
            <label for="geboortedatum" class="col-form-label">
                Geboortedatum
            </label>
            <InputDate id="geboortedatum" @bind-Value=
                "userViewModel.GeboorteDatum" class="form-control" />
        </div>
        <div class="col-2">
            <label for="opleidingscentrum" class="col-form-label">
                Opleidingscentrum
            </label>
            <InputSelect id="opleidingscentrum" class="form-control"
                         @bind-Value="userViewModel.Opleidingscentrum">
                <option>Antwerpen</option>
                <option>Genk</option>
                <option>Heverlee</option>
                <option>Wevelgem</option>
                <option>Wondelgem</option>
            </InputSelect>
        </div>
    </div>
    <div class="row mb-3 ms-1">
        <div class="form-check">
            <InputCheckbox id="rijbewijs" @bind-Value="userViewModel.Rijbewijs"
                       class="form-check-input" />
            <label for="rijbewijs" class="form-check-label">
                Rijbewijs
            </label>
        </div>
    </div>
    <ValidationSummary />
    <button type="submit" class="btn btn-primary mt-2">Submit</button>
</EditForm>
@code {
    private UserViewModel userViewModel = new UserViewModel();
    private void HandleValidSubmit()
    {
        //formulier wordt verwerkt
    }
}
```

- Probeer het formulier opnieuw uit.

  Een mogelijk resultaat :

Opmerking: het datumformaat wordt weergegeven zoals ingesteld in de browsersettings. Intern wordt het formaat “yyyy-mm-dd” gebruikt.

10.3 InputRadio

Blazor beschikt ook over de componenten InputRadioGroup en InputRadio om een groep van radiobuttons weer te geven.

In ons voorbeeld laten we de gebruiker een finaliteitsopleiding kiezen : .NET, Java of PHP.

We beginnen met een enumeration *Finaliteit* toe te voegen.

- Voeg in de folder *Models* onderstaande enumeration toe :

```
public enum Finaliteit
{
    DotNet,
    Java,
    PHP
}
```

- Voeg in de class *UserViewModel* een extra property *Finaliteit* toe :

```
public class UserViewModel
{
    …
    public Finaliteit Finaliteit { get; set; }
}
```

We breiden nu de form uit : we voegen de nodige radiobuttons toe zodat de finaliteit kan gekozen worden.

- Voeg onderstaande vetgedrukte code toe onderaan de form, net voor de validationsummary:

```
@page "/register"
<h1>Registratieformulier</h1>
<EditForm Model="@userViewModel" OnValidSubmit="HandleValidSubmit">
    …
    Finaliteit
    <br />
    <div class="form-check mt-2">
```

```
            <InputRadioGroup @bind-Value="userViewModel.Finaliteit">
                @foreach (var finaliteit in (Finaliteit[])Enum.
                    GetValues(typeof(Finaliteit)))
                {
                    <div class="my-2">
                        <InputRadio Value="finaliteit" class="form-check-input"
                            id="@finaliteit"/>
                        <label class="form-check-label" for="@finaliteit">
                            @finaliteit
                        </label>
                    </div>
                }
            </InputRadioGroup>
        </div>
        <ValidationSummary />
        <button type="submit" class="btn btn-primary mt-2">Submit</button>
    </EditForm>
    …
```

We voegen hier een InputRadioGroup-component toe die we data-binden met de property Finaliteit van de user.

Met een foreach-lus overlopen we alle waarden uit een array van Finaliteit-enumwaarden. Deze array wordt opgevuld met alle mogelijke waarden voor de enum Finaliteit. Voor elke waarde tonen we een *InputRadio* component en een bijhorend label.

- Je kan dit uitproberen.

10.4 InputFile

Met de component InputFile kan de gebruiker een file selecteren op zijn harde schijf. De component InputFile wordt gerenderd naar een input-tag met type *file*.

Als voorbeeld maken we een page waarop we een aantal bestanden kunnen selecteren. Deze bestanden worden dan toegevoegd aan een bestandenlijst.

- Maak in de folder *Pages* een nieuwe component en noem deze *FileSelector.razor.*
- Voeg in *NavMenu* een extra menuitem toe voor de component *FileSelector.* Verwijder eventueel enkele oudere menuitems.

```
…
<div class="nav-item px-3">
    <NavLink class="nav-link" href="file">
        <span class="bi bi-files" aria-hidden="true">
        </span> File Selector
    </NavLink>
</div>
…
```

- Voorzie de component *FileSelector* van onderstaande code:

```
@page "/file"
<h3>Bestandenlijst</h3>
@if (files?.Count > 0) {
<table class="table">
    <thead>
        <tr>
            <th>Naam</th>
            <th>Grootte</th>
```

```
                <th>Laatste wijziging</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var file in files) {
            <tr>
                <td>@file.Name</td>
                <td>@((Math.Ceiling(file.Size / 1024F)).ToString("#.### KB"))</td>
                <td>@file.LastModified.ToString("dd/MM/yyyy HH:mm")</td>
            </tr>
            }
        </tbody>
    </table>
    }
    <InputFile OnChange="@OnInputFileChange" multiple class="btn"/>
    @code {
        private IReadOnlyList<IBrowserFile> files;
        private void OnInputFileChange(InputFileChangeEventArgs e) {
            files = e.GetMultipleFiles();
        }
    }
```

In de code section zien we een membervariabele *files.* Dit is een IReadOnlyList<IBrowserFile> en zal alle bestandsverwijzingen bevatten die we selecteren.

In de layout lijsten we de List<IBrowserFile> uit als deze niet null of leeg is. Een IBrowserFile bevat enkele property's zoals de naam van het bestand waarnaar wordt verwezen (Name), de bestandsgrootte (Size) en het moment waarop het bestand laatst werd gewijzigd (LastModified). De Size zetten we om van bytes naar kilobytes.

Onder de lijst zien we de InputFile component. Deze heeft een Change-event. We koppelen een eventhandler OnInputFileChange aan deze event. Het attribuut multiple geeft aan dat we meerdere bestanden kunnen selecteren.

In de eventhandler OnInputFileChange kan je de gegevens van de geselecteerde bestanden opvragen via de method *GetMultipleFiles()* van de parameter e. Het resultaat van deze method slaan we op in de membervariabele *files*.

Je kan er ook voor kiezen om de membervariabele *files* het type List<IBrouwerFile> te geven. Je zal dan het resultaat van de method *GetMultipleFiles()* nog moeten converteren naar een List met de method ToList().

- Je kan de page uitproberen.

Een mogelijk resultaat :

Bestandenlijst

Naam	Grootte	Laatste wijziging
ASP.NET Core 6 Blazor.docx	2705 KB	28/01/2022 13:11
Blazor.docx	422 KB	19/05/2021 07:33
changelog20220131.txt	1 KB	28/01/2022 13:27

Bestanden kiezen 3 bestanden

10.5 Een eigen validation attribute

Via de data-annotations hebben we al heel wat mogelijkheden op vlak van validatie. Wil je toch een validatie toevoegen die niet is voorzien dan kan je een eigen validator aanmaken.

Als voorbeeld maken we een validator die oplegt dat de geboortedatum in het verleden moet liggen. De werkwijze is identiek als in ASP.NET Core.

- Voeg, bijvoorbeeld in een nieuwe folder *Validators*, een nieuwe class *GeboortedatumValidator.cs* toe.
- Neem onderstaande code over in *GeboortedatumValidator.cs* :

```
using System.ComponentModel.DataAnnotations;
…
public class GeboortedatumValidator : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        if (value == null)
            return true;
        if (!(value is DateTime))
            return false;
        return ((DateTime) value) < DateTime.Today;
    }
}
```

- Pas de custom validator nu toe op het veld *Geboortedatum* in *UserViewModel.cs*. Voeg eventueel het nodige using-statement toe indien VS dit niet zelf doet.

```
[GeboortedatumValidator(ErrorMessage =
    "Geboortedatum moet in het verleden liggen")]
public DateTime GeboorteDatum { get; set; }
```

- Probeer uit : start de application en browse naar /register. Vul een datum in de toekomst in en je krijgt meteen een foutboodschap.

Opmerking : er is ook een variant met een *IsValid*-method die een ValidationResult teruggeeft. De code is dan als volgt :

```
protected override ValidationResult IsValid(object value,
    ValidationContext validationContext)
{
    if ((DateTime)value < DateTime.Now)
        return null;
    return new ValidationResult(ErrorMessage,
        new[] { validationContext.MemberName });
}
```

10.6 Comparevalidator

Om twee velden met elkaar te vergelijken gebruiken we een [Compare]-annotation.

Een voorbeeld : we voegen de velden Email en ConfirmEmail toe. Die moeten gelijk zijn.

- Voeg in *UserViewModel.cs* de property's Email en ConfirmEmail toe :

```
public class UserViewModel
{
    …
    public string Email { get; set; }
    [Compare("Email", ErrorMessage = "Email en ConfirmEmail zijn verschillend")]
    public string ConfirmEmail { get; set; }
}
```

- Voeg in *Register.razor* extra inputvelden toe voor deze property's :

```
…
<div class="row my-3">
    <div class="col-3">
        <label for="email" class="col-form-label">Email</label>
        <InputText id="email" @bind-Value="userViewModel.Email"
                   class="form-control" />
    </div>
    <div class="col-3">
        <label for="confirmemail" class="col-form-label">
            ConfirmEmail
        </label>
        <InputText id="confirmemail" @bind-Value="userViewModel.ConfirmEmail"
                   class="form-control" />
    </div>
</div>
…
```

- Probeer uit.

10.7 Validatie van complex types en collection types

De validatie via data annotations werkt enkel voor primitive types. Voor complex types en collection types gebruik je een ObjectGraphDataAnnotationsValidator uit de NuGet package *Microsoft.AspNetCore.Components.DataAnnotations.Validation*.

- Installeer de NuGet package *Microsoft.AspNetCore.Components.DataAnnotations.Validation.* Opgepast, dit is een prerelease. Vink dus de optie *Include prerelease* aan op het tabblad *Browse* bij het zoeken naar deze NuGet package.

Als voorbeeld voegen we een property *Adres* toe aan de class UserViewModel. Deze property is van het type *Adres* dat bestaat uit de string property's *StraatEnNr*, *Postcode* en *Gemeente*.

- Voeg in de folder *Models* onderstaande class *Adres* toe :

```
public class Adres
{
    [Required(ErrorMessage = "Straat en nummer is een verplicht veld")]
    public string StraatEnNr { get; set; }
    [Required(ErrorMessage = "{0} is een verplicht veld")]
    public string Postcode { get; set; }
    [Required(ErrorMessage = "{0} is een verplicht veld")]
    public string Gemeente { get; set; }
}
```

In deze class maken we van alle property's een verplicht veld via de annotation [Required] uit de namespace *System.ComponentModel.DataAnnotations*.

In de class UserViewModel voegen we nu een Adres-veld toe.

- Wijzig de code in *UserViewModel.cs* als volgt :

```
public class UserViewModel
{
    …
    [ValidateComplexType]
    public Adres Adres { get; set; } = new Adres();
}
```

Je voorziet de extra property van een annotation ValidateComplexType.

In de component *Register.razor* voeg je 3 extra inputvelden toe.

- Voeg onderstaande code toe in *Register.razor* :

```
…
<div class="row my-3 ">
    <div class="col-3">
        <label for="adres" class="col-form-label">
            Straat en huisnummer
        </label>
        <InputText id="adres" @bind-Value="user.Adres.StraatEnNr"
            class="form-control" />
    </div>
    <div class="col-1">
        <label for="postcode" class="col-form-label">
            Postcode
        </label>
        <InputText id="postcode" @bind-Value="user.Adres.Postcode"
                class="form-control"/>
    </div>
    <div class="col-2">
        <label for="gemeente" class="col-form-label">
            Gemeente
        </label>
        <InputText id="gemeente" @bind-Value="user.Adres.Gemeente"
                class="form-control"/>
    </div>
</div>
…
```

- Tenslotte vervangen we in de EditForm de component DataAnnotationsValidator door een ObjectGraphDataAnnotationsValidator. Wijzig de code in Register.razor als volgt :

```
…
<EditForm Model="@userViewModel" OnValidSubmit="HandleValidSubmit">
    <ObjectGraphDataAnnotationsValidator />
    …
```

- Probeer opnieuw uit, de extra velden worden gevalideerd.

10.8 EditContext

Voor de meeste toepassingen is bovenstaande werkwijze om een form te maken voldoende. Je kan echter in een form ook gebruik maken van een EditContext voor extra validatiemogelijkheden.
Een EditContext houdt bij welke velden er gewijzigd zijn en ook wat de validatiestatus is van het formulier: welke velden zijn (niet) geldig, welke foutboodschappen zijn er,...

Een eenvoudig voorbeeld : beschouw onderstaande form waarin je de naam van jouw huisdier kan intikken.

```
<EditForm EditContext="editContext" OnSubmit="HandleHuisdierForm">
    <label>Naam van jouw huisdier : </label>
    <InputText @bind-Value="huisdier.Name"/>
    <button type="submit">Submit</button>
</EditForm>
@code {
    private EditContext editContext;
    private Huisdier huisdier = new Huisdier() { HuisdierId = 1, Naam = "Max" };
    protected override void OnInitialized()
    {
        editContext = new EditContext(huisdier);
    }
    private void HandleHuisdierForm()
    {
        if (editContext.Validate())
            Console.WriteLine(huisdier.Naam);
        else
            Console.WriteLine("Not valid");
    }
}
```

In de layout merk je dat EditForm deze keer geen Model-attribuut heeft maar wel een EditContext. Deze EditContext wordt in *OnInitialized()* geïnitialiseerd : het model (het object huisdier) wordt via de de constructor meegegeven. Het model wordt dus niet meer rechtstreeks doorgegeven aan de EditForm maar via een EditContext-object.

Een submit van de form activeert een method die via de Validate()-method van de EditContext controleert of de form geldig valideert.

Tot hiertoe hebben we geen extra mogelijkheden gebruikt t.o.v. de vorige werkwijze (zonder EditContext). Wanneer we naar de definitie van de EditContext class kijken zien we een aantal interessante events en methods :

```
public sealed class EditContext
{
    public EditContext(object model);
    public object Model { get; }
    public event EventHandler<FieldChangedEventArgs> OnFieldChanged;
    public event EventHandler<ValidationRequestedEventArgs> OnValidationRequested;
    public event EventHandler<ValidationStateChangedEventArgs> OnValidationStateChanged;
    public FieldIdentifier Field(string fieldName);
    public IEnumerable<string> GetValidationMessages();
    public IEnumerable<string> GetValidationMessages(FieldIdentifier fieldIdentifier);
    public bool IsModified();
    public bool IsModified(in FieldIdentifier fieldIdentifier);
```

```
    public void MarkAsUnmodified(in FieldIdentifier fieldIdentifier);
    public void MarkAsUnmodified();
    public void NotifyFieldChanged(in FieldIdentifier fieldIdentifier);
    public void NotifyValidationStateChanged();
    public bool Validate();
}
```

De readonly property **Model** kan worden ingevuld via de constructor **EditContext()**.

Er zijn 3 events : **OnFieldChanged**, **OnValidationRequested** en **OnValidationStateChanged**. Deze namen spreken voor zich : het zijn events die optreden wanneer de inhoud van een veld is gewijzigd, wanneer werd gevraagd de geldigheid van de form te controleren en wanneer de validatiestatus is veranderd.

In de methods zien we dat er vaak een **FieldIdentifier** wordt gebruikt. Dit is een class waarmee we een bepaald veld aanduiden. Het heeft een property FieldName voor de naam van het veld en een property Model dat je het model oplevert waartoe het veld behoort.

De method **Field(fieldname)** levert de FieldIdentifier op die hoort bij de opgegeven fieldname.

De methods **GetValidationMessages()** en **GetValidationMessages(fieldidentifier)** leveren ofwel álle validationmessages op ofwel deze voor één bepaalde fieldidentifier.

IsModified() geeft aan of de editcontext is gewijzigd, **IsModified(fieldidentifier)** geeft aan of één welbepaald veld is gewijzigd.

Eén veld of alle velden kunnen gemarkeerd worden als ongewijzigd met de methods **MarkAsUnmodified(fieldidentifier)** en **MarkAsUnmodified()**.

NotifyFieldChanged triggert de OnFieldChanged event voor een bepaald veld.
NotifyValidationStateChanged signaleert dat de validatiestatus gewijzigd is.

Validate() voert de validatiecontrole door.

In onderstaand voorbeeld proberen we één en ander uit. Door gebruik te maken van de extra mogelijkheden die EditContext biedt zullen we de submit-knop pas actief maken wanneer alle velden geldig valideren.

- Maak in de folder *Pages* een nieuwe component : *Gastenboek.razor*.
- Voeg in *NavMenu* een extra menuitem toe dat verwijst naar deze component :

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="gastenboek">
        <span class="oi oi-stickies" aria-hidden="true"></span>
            Gastenboek
    </NavLink>
</div>
```

We kunnen een boodschap achterlaten in het gastenboek door een naam en een tekst te noteren. Daarvoor maken we een nieuwe class : *Boodschap*.

- Voeg in de folder *Models* onderstaande class *Boodschap* toe :

```
public class Boodschap
{
    [Required]
    [MaxLength(20)]
    public string Naam { get; set; }
```

```
    [Required]
    public string Tekst { get; set; }
}
```

- Vervang de inhoud van *Gastenboek.razor* door onderstaande code :

```
@page "/gastenboek"
<h1>Mijn gastenboek</h1>
<h4>Laat hier jouw boodschap achter</h4>
<EditForm EditContext="@editContext">
    <DataAnnotationsValidator />
    <label for="naam" class="col-form-label">Naam: </label>
    <InputText id="naam" class="form-control" @bind-Value="@Model.Naam"/>
    <ValidationMessage For="@(() => Model.Naam)" />
    <label for="tekst" class="col-form-label">Tekst: </label>
    <InputTextArea id="tekst" class="form-control" @bind-Value="@Model.Tekst"/>
    <ValidationMessage For="@(() => Model.Tekst)" />
    <button type="submit" class="btn btn-primary mt-2"
        disabled="@OkKnopDisabled">Ok</button>
</EditForm>

@code
{
    private EditContext editContext;
    private Boodschap Model = new Boodschap();

    protected string OkKnopDisabled { get; set; } = "disabled";

    protected override void OnInitialized()
    {
        editContext = new EditContext(Model);
        editContext.OnFieldChanged += editContext_OnFieldChanged;
        base.OnInitialized();
    }

    protected override void OnAfterRender(bool firstRender)
    {
        base.OnAfterRender(firstRender);
        if (firstRender)
            SetOkDisabledStatus();
    }

    private void editContext_OnFieldChanged(object sender,
        FieldChangedEventArgs e)
    {
        SetOkDisabledStatus();
    }

    private void SetOkDisabledStatus()
    {
        if (editContext.Validate())
            OkKnopDisabled = null;
        else
            OkKnopDisabled = "disabled";
    }

    public void Dispose()
    {
        editContext.OnFieldChanged -= editContext_OnFieldChanged;
    }
}
```

Een woordje uitleg bij de code van deze component.

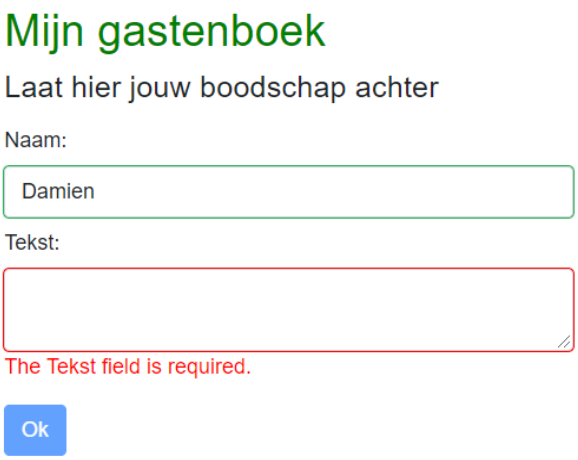
In de layout merken we weinig nieuws. De EditForm gebruikt deze keer een EditContext en de inputvelden worden gebonden aan de property's *Naam* en *Tekst* van het Model uit deze EditContext. Voor de afwisseling gebruiken we geen validationsummary maar beelden we eventuele validation-messages af onder het inputveld. De submitbutton heeft een attribuut *disabled* dat zijn waarde krijgt aangereikt vanuit de property *OkKnopDisabled.* Het is de bedoeling dat de OK-knop pas bruikbaar zal zijn wanneer beide velden geldig valideren.

In de code-section wordt de EditContext geïnitialiseerd in de method *OnInitialized().* In diezelfde method koppelen we de eventhandler *editContext_OnFieldChanged* aan de event *OnFieldChanged* van de EditContext. Diezelfde eventhandler 'ontkoppelen' we opnieuw in de *Dispose()*-method. De property *OkKnopDisabled* krijgt bij aanvang de waarde *disabled* zodat de knop aanvankelijk niet kan gebruikt worden.

Zowel bij een eerste weergave van de page als bij elke wijziging moeten we controleren of er in beide velden iets is ingevuld en eventueel de status van de OK-knop wijzigen. We roepen daarvoor in de method *OnAfterRender()* en in de eventhandler *editContext_OnFieldChanged()* de method *SetOkDisabledStatus()* op.

In *SetOkDisabledStatus()* valideren we de form. Bij een positief resultaat wordt de property *OkKnopDisabled* null waardoor het attribuut *disabled* niet wordt gerenderd. Is het resultaat negatief dan krijgt de property de waarde *disabled.*

- Je kan dit uitproberen.



Zolang niet beide velden ingevuld zijn is de OK-knop disabled. De knop heeft een lichtere blauwe tint. Is alles ingevuld dan krijgt de knop een donkerblauwe tint.

10.9 Oefening

Ontwerp onderstaand sollicitatieformulier :

Gebruik een EditForm. Kies zelf of je het model aangeeft via een Model-attribuut of met EditContext.

De velden in het formulier hebben volgende eigenschappen :

Naam en voornaam : string, veld is verplicht in te vullen en max. 30 tekens lang.

Geboortedatum : datetime, moet in het verleden liggen.

Emailadres : string, te valideren als een emailadres en verplicht in te vullen

Rijbewijs : boolean

Hoogste diploma : enum met als mogelijke waarden *geen*, *middelbaaronderwijs*, *hogeronderwijsbuitenuniv* en *hogeronderwijsuniv*. Gebruik radiobuttons.

Functie : string met als mogelijke waarden *software tester*, *junior .NET ontwikkelaar*, *full stack .NET developer*, *medior/senior developer.* Gebruik een keuzelijst.

Tip : enums kunnen geen spaties bevatten. Om het label van de opties voor het veld *Diploma* toch te kunnen afbeelden mét spaties kunnen we de beschrijving ophalen uit een List<string>.

Je voegt in de code section een static readonly List<string> toe :

```
private static readonly List<string> diplomas = new List<string> { "Geen", "Middelbaar
onderwijs", "Hoger onderwijs, niet universitair", "Hoger onderwijs, universitair" };
```

Je beeldt het label voor een radiobutton dan als volgt af :

```
<label class="form-check-label" for="@diploma">@diplomas[(int)diploma]</label>
```

Hierbij is *diploma* de lusvariabele in de foreach-lus die alle enumwaarden van de enum Diploma overloopt. Als we *diploma* als index willen gebruiken dan moeten we deze converteren naar een int.

11 Routing

In dit hoofdstuk bekijken we het Blazor routing mechanisme in detail.

11.1 De Router component

In een Blazor app wordt de routing uitgevoerd in de component *Routes.razor*. Deze component wordt in App.razor in de body opgeroepen. De component Routes heeft standaard onderstaande inhoud :

```
<Router AppAssembly="typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="routeData" DefaultLayout="typeof(MainLayout)" />
        <FocusOnNavigate RouteData="routeData" Selector="h1" />
    </Found>
</Router>
```

Deze root-component bevat een component *Router.* Dit is een templated component waarvan hier enkel de parameter *Found* wordt gebruikt. De router zoekt in de componenten van de assembly, aangeduid met de parameter *AppAssembly* of er een component is met een route die overeenkomt met de route template uit de request.

Is dit het geval dan geeft een RouteView component de component weer die beantwoordt aan de request. Standaard wordt ook een default layout meegegeven. Dit is niet verplicht. De component FocusOnNavigate zorgt ervoor dat na het navigeren naar de gevonden pagina de tag die bij Selector staat vermeld de focus krijgt. In dit geval is dit de header h1. Dit is nuttig voor mensen met een beperking die een screen reader gebruiken. De headertekst wordt dan voorgelezen zodat de gebruiker weet naar welke pagina er genavigeerd is.

11.2 Routing naar een component uit een Razor class library

Je kan de router ook routetemplates van componenten uit een Razor classlibrary laten afhandelen. Een Razor Class Library is een library waarin je componenten, static files en code kan opslaan die je wil hergebruiken in verschillende applicaties.

Om een component uit een Razor class library te kunnen aanspreken moeten er een aantal stappen worden gezet. Welke stappen dit zijn hangt af van de gekozen rendermode van de Blazor applicatie.

Stap 1 : Om te kunnen verwijzen naar een component uit een Razor class library moet er een reference gelegd worden naar de class library. Bestaat jouw Blazor app uit twee projecten dan leg je de reference vanuit elk project dat de library gebruikt.

Stap 2 : In het bestand Program.cs voeg je een extra assembly toe door een oproep van de method AddAdditionalAssemblies(). Bestaat jouw Blazor app uit twee projecten dan voeg je de code toe in Program.cs van het project zonder .Client-extensie.

Stap 3 : Gebruikt jouw app de rendermode Static Server of Interactive Server (dus niet bij WebAssembly en Auto) dan is er nog een derde stap nodig. Je voegt in Routes.razor de extra assembly toe via het attribuut AdditionalAssemblies.

We proberen dit uit.

We voegen aan de solution eerst een Razor Class Library toe.

- Klik met de rechtermuistoets op de solution en kies *Add – New Project…*
- Kies een *Razor Class Library* en klik *Next.*
- Geef het project de naam *RazorClassLibrary* en klik op *Next*.
- Kies .NET 8.0, vink de optie *Support pages and views* **UIT** en kies *Create*.

De nieuwe razor library bevat reeds een component met de naam *Component1*. We zorgen er nu voor dat deze component vanuit onze app kan aangesproken worden via de route */uitlibrary*.

- Open de component *Component1* en voeg er de route /*uitlibrary* aan toe :

```
@page "/uitlibrary"
<div class="my-component">
    This component is defined…
</div>
```

11.2.1 De class library gebruiken vanuit een Interactive Server Blazor app

We proberen de component *Component1* uit de Razor class library nu te gebruiken vanuit een Interactive Server Blazor app. In paragraaf *2.4 Blazor app templates* heb je een dergelijke app met de naam *BlazorServer* aangemaakt.

Om de component uit de classlibrary te kunnen gebruiken moeten we eerst een reference leggen naar de library.

- Klik in de Server Explorer met de rechtermuistoets op de naam van het project BlazorServer en kies *Add – Project Reference.*
- Zet een vinkje naast de class library en klik op OK.

In een volgende stap voegen we de nodige code toe in Program.cs zodat de assembly waarin de extra component is gedefinieerd wordt toegevoegd.

- Wijzig in het project BlazorServer de code in Program.cs als volgt :

```
…
app.MapRazorComponents<App>()
   .AddInteractiveServerRenderMode()
   .AddAdditionalAssemblies(typeof(RazorClassLibrary.Component1).Assembly);
…
```

Je voegt de assembly toe door een oproep van de method AddAdditionalAssemblies. Deze method krijgt als parameter de assembly mee waarin de componentclass Component1 is gedefinieerd. Gebruik je ook andere componenten uit die extra assembly dan volstaat het van deze method één keer toe te voegen.

In een Blazor Server app moeten we ook in Routes.razor de extra assembly toevoegen.

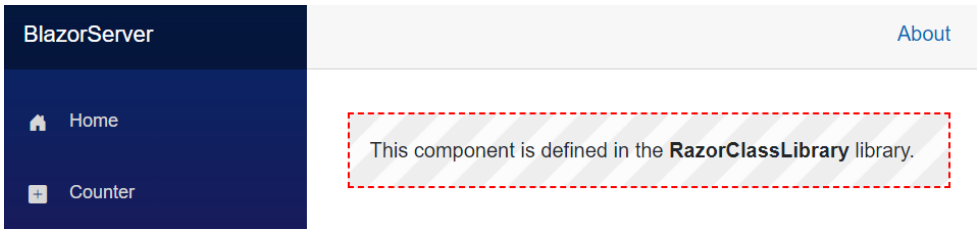
- Wijzig de code in Routes.razor als volgt :

```
<Router AppAssembly="typeof(Program).Assembly"
  @rendermode="InteractiveServer"
  AdditionalAssemblies="new[] {
    typeof(RazorClassLibrary.Component1).Assembly }" >
```

We kunnen nu de component Component1 benaderen vanuit de Blazor Server app.

- Stel het project BlazorServer in als opstartproject.

- Start de Blazor app en browse naar */uitlibrary*. Je ziet de standaardtekst uit *Component1* :



11.2.2 De class library gebruiken vanuit een WebAssembly Blazor app

Ook in een WebAssembly app leggen we eerst een extra reference naar de Razor class library. Je doet dit enkel vanuit het opstartproject.

- Leg vanuit het project BlazorWebAssembly een reference naar de Razor class library.
- Leg ook vanuit het project BlazorWebAssembly.Client een reference naar die library.

Je voegt de Razor class library assembly ook toe in de Program.cs in het opstartproject.

- Wijzig in het project BlazorWebAssembly de code in Program.cs als volgt :

```
…
app.MapRazorComponents<App>()
    .AddInteractiveWebAssemblyRenderMode()
    .AddAdditionalAssemblies(typeof(BlazorWebAssembly.Client._Imports).Assembly)
    .AddAdditionalAssemblies(typeof(RazorClassLibrary.Component1).Assembly);
…
```

Je voegt in het project BlazorWebAssembly.Client in Routes.razor de assembly toe.

- Wijzig de code in Routes.razor als volgt :

```
<Router AppAssembly="typeof(Program).Assembly"
  @rendermode="InteractiveWebAssembly"
  AdditionalAssemblies="new[] {
    typeof(RazorClassLibrary.Component1).Assembly }" >
```

Je kan de app nu uitproberen.

- Stel het project BlazorWebAssembly in als opstartproject.
- Start de app en browse naar */uitlibrary*. Je krijgt de standaardtekst uit *Component1* te zien.

11.2.3 De class library gebruiken vanuit een Static Server Blazor app

In een Static Server app leggen we een reference naar de Razor class library en voegen we de assembly toe in Program.cs.

- Leg vanuit het project BlazorStatic een reference naar de Razor class library.

Je voegt in Program.cs de Razor class library assembly toe.

- Wijzig in het project BlazorStatic de code in Program.cs als volgt :

```
…
app.MapRazorComponents<App>()
    .AddAdditionalAssemblies(typeof(RazorClassLibrary.Component1).Assembly);
…
```

Je kan de component nu oproepen vanuit het opstartproject.

- Stel het project BlazorStatic in als opstartproject.
- Start de Blazor app en browse naar */uitlibrary*. Je krijgt de standaardtekst uit *Component1* te zien.

11.2.4 De class library gebruiken vanuit een Interactive Auto Blazor app

De werkwijze in een Interactive Auto Blazor app is vrij gelijkaardig aan de werkwijze in een Interactive Server app.

Je start opnieuw met het leggen van references naar de Razor class library. Je doet dit zowel in het opstartproject als in het project met extra extensie .Client.

- Leg vanuit het project BlazorAuto een reference naar de Razor class library.
- Leg ook vanuit het project BlazorAuto.Client een reference naar die library.

In het opstartproject voeg je in Program.cs een extra assembly toe.

- Wijzig in het project BlazorAuto de code in Program.cs als volgt :

```
…
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode()
    .AddInteractiveWebAssemblyRenderMode()
    .AddAdditionalAssemblies(typeof(BlazorAuto.Client._Imports).Assembly)
    .AddAdditionalAssemblies(typeof(RazorClassLibrary.Component1).Assembly);
…
```

Je voegt in het project BlazorAuto.Client in Routes.razor de assembly toe.

- Wijzig de code in Routes.razor als volgt :

```
<Router AppAssembly="typeof(Program).Assembly"
  @rendermode="InteractiveAuto"
  AdditionalAssemblies="new[] {
    typeof(RazorClassLibrary.Component1).Assembly }" >
```

We kunnen opnieuw uitproberen.

- Stel het project BlazorAuto in als opstartproject.
- Start de app en browse naar */uitlibrary*. Je krijgt de standaardtekst uit *Component1* te zien.

11.3 NavLink

Je gebruikt de component *NavLink* om een link naar een component te laten renderen. De NavLink component hebben we al enkele keren gebruikt in de NavMenu component.

Misschien heb je al gemerkt dat wanneer we een menuitem kiezen in NavMenu, dit menuitem een afwijkende opmaak krijgt. Ook wanneer we in de browser een url intikken die matcht met een href van één van de menuitems dan wordt dat menuitem 'actief'. We krijgen deze feature gratis en voor niks aangeboden door gebruik te maken van een NavLink.

Dus wanneer we browsen naar */weather* dan wordt het menuitem *Weather* actief. En wanneer we browsen naar */weather/3* ? Ook dan wordt dit menuitem actief want een NavLink heeft een *Match*

attribuut waarmee je een *NavLinkMatch* option kan instellen. De mogelijke waarden voor deze NavLinkMatch option zijn *NavLinkMatch.All* en *NavLinkMatch.Prefix*.

De option *NavLinkMatch.Prefix* is de standaardwaarde en zorgt ervoor dat de link actief is van zodra de URL in de browser begint met de waarde uit het href-attribuut. Dus of je nu browset naar /weather of /weather/3, het menuitem *Weather* zal altijd actief zijn.

In NavMenu heeft enkel de menuoptie *Home* een Match-attribuut met waarde *NavLinkMatch.All*. Dit betekent dat de URL in de browser exact moet overeenkomen met de waarde van het href-attribuut. Het menuitem *Home* zal dus enkel actief zijn wanneer je browset naar /.

Bij wijze van oefening herstellen we een foutje in het menuitem Hobbies. In de paragraaf over geneste layouts maakten we een submenu met de items Books, Pictures en Music. Wanneer je in het hoofdmenu de optie *Hobbies* kiest wordt de optie Books actief én ook het menuitem *Hobbies.* Wanneer je echter *Pictures* of *Music* kiest dan is het menuitem *Hobbies* niet langer actief.

We lossen dit op door de routes in *Books, Pictures* en *Music* te veranderen én door de optie *NavLinkMatch* te gebruiken.

We passen eerst de routes van de componenten *Books, Pictures* en *Music* aan.

- Wijzig de route in *Books.razor* als volgt :

```
@page "/hobbies"
```

- Wijzig de route in *Pictures.razor* naar /hobbies/pictures :

```
@page "/hobbies/pictures"
```

- Wijzig de route in *Music.razor* naar /hobbies/pictures :

```
@page "/hobbies/music"
```

- Je wijzigt nu in *NavMenu.razor* de NavLink in het Hobbies menuitem als volgt :

```
<NavLink class="nav-link" href="hobbies">
    <span class="bi bi-book" aria-hidden="true"></span> Hobbies
</NavLink>
```

Je verwijst met de reference *hobbies* naar de page *Books.razor.* De NavLink zal echter niet alleen actief zijn voor deze page maar ook voor Pictures.razor én voor Music.razor omdat voor een NavLink by default het Match-attribuut op NavLinkMatch.Prefix is gezet. De routes van Pictures.razor én van Music.razor beginnen immers ook met *hobbies.*

Tenslotte breng je ook nog enkele wijzigingen aan in *SubLayout.razor.*

- Wijzig de NavLink's in *SubLayout.razor* als volgt :

```
<li class="nav-item px-3">
    <NavLink class="nav-link" href="/hobbies" Match="NavLinkMatch.All" >
        <span class="bi bi-book" aria-hidden="true"></span> Books
    </NavLink>
</li>
<li class="nav-item px-3">
    <NavLink class="nav-link" href="/hobbies/pictures" >
        <span class="bi bi-images" aria-hidden="true"></span> Pictures
    </NavLink>
</li>
<li class="nav-item px-3">
    <NavLink class="nav-link" href="/hobbies/music" >
```

```
            <span class="bi bi-music-note-beamed" aria-hidden="true"></span> Music
        </NavLink>
    </li>
```

Je wijzigt dus de href-attribuutwaarden naar de routes van de diverse hobbypages. Bij *Books* zet je het attribuut *Match* op *NavLinkMatch.All*. Doe je dit niet dan is deze menuoptie ook actief wanneer je voor *Music* of *Pictures* kiest omdat *NavLinkMatch.Prefix* de defaultwaarde is voor *Match*.

- Je kan opnieuw uitproberen. Na selectie van het hoofdmenuitem *Hobbies* wordt de subpage *Books* zichtbaar. De tab van deze page is actief. Dat van de andere tabs niet. Wissel je naar een andere tab dan wordt deze actief en de andere niet.

11.4 NavigationManager

Om vanuit C#-code te navigeren naar een andere URL gebruiken we een NavigationManager.

We proberen de voornaamste property's en methods van NavigationManager uit aan de hand van een voorbeeld.

- Voeg in de folder Pages een nieuwe component toe en noem deze *Navman.razor*.
- Voeg er onderstaande code aan toe :

```
@page "/navman"
@inject NavigationManager navmanager
<h3>Navman</h3>
<input @bind-value="URL" />
<button @onclick="Navigeer" class="btn btn-primary">Navigeer</button>
<p>Huidige URI : @uri</p>
<p>Huidige BaseURI : @baseuri</p>
<p>Naar een absolute URI : @absuri</p>
<p>Naar een BaseRelativePath : @baserelpath</p>
@code {
    public string URL { get; set; } = "weather";
    string uri;
    string baseuri;
    string absuri;
    string baserelpath;
    void Navigeer() {
        navmanager.NavigateTo(URL);
    }
    protected override void OnInitialized()
    {
        uri = navmanager.Uri;
        baseuri = navmanager.BaseUri;
        absuri = navmanager.ToAbsoluteUri(URL).ToString();
        baserelpath = navmanager.ToBaseRelativePath(absuri);
    }
}
```

Bovenaan de code injecteren we een NavigationManager-object dat we *navmanager* noemen. Via dit object zullen we enkele property's en methods van de NavigationManager uitproberen.

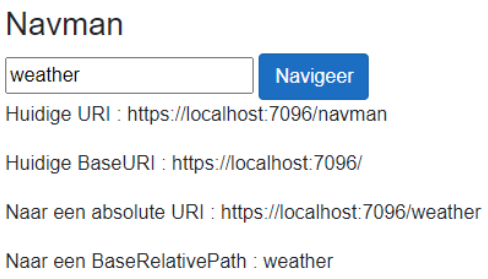
Via een inputveld kunnen we een URL intikken. Dit inputveld is via databinding gelinkt aan de variabele URL. Een klik op de Navigeer-button roept de Navigeer-method op uit de code-section. In die method gebruiken we **NavigateTo()**-method om te navigeren naar de ingetikte URL.

In de method *OnInitialized()* halen we twee property's op : **Uri** en **BaseUri**. Uri levert de huidige, volledige URI op. BaseUri levert de basis URI op, dus zonder het pad naar de gewenste component.

Het inputveld bevat standaard de route naar de fetchdata component. Om de absolute URI te verkrijgen naar deze component gebruiken we de NavigationManager-method ***ToAbsoluteUri()***. Omgekeerd gebruiken we de method *ToBaseRelativePath()* om vertrekkend vanaf een absolute URI een relatieve URI te verkrijgen.

- Probeer uit door te browsen naar */navman*.

Het resultaat :



- Tik eventueel een andere route (bvb. *blog*) in het inputveld en klik op *Navigeer*.

12 JavaScript interoperability

In een Blazor app kan je JavaScript functions oproepen vanuit .NET methods en .NET methods vanuit JavaScript functions. We noemen dit *JavaScript interoperability* of kortweg *JS interop*.

12.1 JavaScript oproepen vanuit C#

Om vanuit Blazor JavaScript-code uit te voeren gebruiken we een IJSRuntime service. Die moet je eerst injecteren in de component. Daarna gebruik je op die IJSRuntime de *InvokeAsync<T>*-method om de JavaScript-function op te roepen. T is hier het returntype. Je geeft de function identifier mee van de function die je wil oproepen, samen met eventuele parameters. Zowel de parameters als het returntype moeten serialiseerbaar zijn.

Je moet wel oppassen wanneer je de function oproept. Een Blazor app kan geprerenderd worden. Dit is een soort voorbereiding van de rendering van een page. Tijdens dit proces kan je geen JavaScript-calls uitvoeren. Het is sowieso beter een JavaScript-function pas uit te voeren wanneer het renderingproces volledig klaar is. Wil je een JavaScript-function automatisch laten oproepen, doe dat dan pas ten vroegste in de OnAfterRendering()-method. Dan ben je zeker dat de DOM volledig is samengesteld.

Een eerste voorbeeld : we maken een JavaScript-function *kwadraat()* die het kwadraat berekent van een getal. We gebruiken deze function vervolgens in een Blazor component.

Om te beginnen nemen we de JavaScript-function op in *App.razor*.

- Voeg onderstaand script toe in het <head>-element in *App.razor* :

```
…
<head>
    …
    <script>
        window.kwadraat = (getal) => {
            return getal*getal;
        };
    </script>
</head>
…
```

Je kan er uiteraard ook voor kiezen het script op te nemen in een afzonderlijk bestand :

Bijvoorbeeld : `<script src="JsInterop.js"></script>`

We roepen deze JavaScript-function nu op vanuit een Blazor component.

- Voeg in de folder *Pages* een nieuwe Blazor component toe en noem deze *JSInterop.razor*.
- Voeg in NavMenu een extra menuitem toe :

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="JSInterop">
        <span class="oi oi-vector-pen" aria-hidden="true"></span> JS Interop
    </NavLink>
</div>
```

- Zet eventueel enkele oudere menuitems in commentaar of wis ze.

- Voeg onderstaande code toe in *JSInterop.razor* :

```
@page "/JSInterop"
@inject IJSRuntime JSRuntime;
<h1>JSInterop</h1>
<span for="getal">Getal: </span>
<input id="getal" type="number" @bind-value="@Getal" />
<button type="button" class="btn btn-primary ms-2" @onclick="BerekenKwadraat">
    Bereken kwadraat
</button>
<p class="mt-2 h3">
    <span class="badge bg-success p-3">
        Het kwadraat van @Getal is @hetKwadraat.
    </span>
</p>

@code {
    public int Getal { get; set; }
    private int hetKwadraat = 0;
    private async Task BerekenKwadraat()
    {
        hetKwadraat =
            await JSRuntime.InvokeAsync<int>("kwadraat", Getal);
    }
}
```

Bovenaan de code injecteren we een IJSRuntime. We geven dit object de naam JSRuntime. Deze naam is uiteraard vrij te kiezen. We gebruiken deze service in de method *BerekenKwadraat.*

Via een inputveld laten we een getal invullen. Een klik op een button voert de method *BerekenKwadraat* uit. Daaronder voeren we het resultaat uit van deze method.

In de method *BerekenKwadraat* roepen we de IJSRuntime-method *InvokeAsync* uit. De eerste parameter is de naam van de JavaScript-function die we willen uitvoeren, de tweede parameter is de parameter voor deze function. Het resultaat van de functionoproep is een int.

- Probeer uit. Start de applicatie en kies het menuitem *JS Interop*.
- Tik een getal in en klik op de knop *Bereken kwadraat.* De JavaScript-code wordt uitgevoerd.

In een tweede voorbeeld vragen we de gebruiker zijn naam in een popupvenster in te tikken. We geven deze naam vervolgens weer in de page door de DOM te manipuleren. We laten het script deze keer automatisch uitvoeren van zodra de Blazor page wordt opgevraagd.

- Voeg onderstaande code toe in de script-tag in *App.razor* :

```
window.userInputFunctions = {
    vraagNaam: function (text) {
        return prompt(text, 'Tik hier jouw naam');
    },
    toonBoodschap: function (boodschap) {
        document.getElementById('welkomstTekst').innerText = boodschap;
    }
};
```

We groeperen deze keer twee functions in een *userInputFunctions*-object. Wanneer we dit doen dan roep je de function *vraagNaam* op via *userInputFunctions.vraagNaam.*

De function *vraagNaam* heeft een parameter *text*. Deze parameter bevat de tekst die moet weergegeven worden in het prompt-venster. De prompt-function heeft 2 parameters : de tekst die moet getoond worden en een defaulttekst voor het inputveld. *vraagNaam* retourneert de ingevulde tekst.

De tweede function, *toonBoodschap*, zoekt een element op met de id *welkomstTekst* en vervangt de innerText van dit element door de meegegeven parameter *boodschap*.

We gebruiken beide functions nu in de component *JSInterop.razor.*

- Om de welkomsttekst te kunnen weergeven voeg je in *JSInterop.razor* net onder de <h1>-tag volgende layout-code toe :

```
<h3 id="welkomstTekst"/>
```

De JavaScript-code laten we uitvoeren nadat de page werd gerenderd, dus in de method *OnAfterRenderAsync()*.

- Voeg in de code-section van JSInterop.razor onderstaande code toe :

```
protected async override Task OnAfterRenderAsync(bool firstRender)
{
    if (firstRender) {
        var naam = await JSRuntime.InvokeAsync<string>(
            "userInputFunctions.vraagNaam",
            "Jouw naam: ");
        await JSRuntime.InvokeVoidAsync(
            "userInputFunctions.toonBoodschap",
            $"Hallo {naam}! Welkom bij Blazor!");
    }
}
```

In de bovenstaande code gebruiken we de IJSRuntime-method *InvokeAsync()* om de JavaScript-function *userInputFunctions.vraagNaam()* uit te voeren. Het resultaat van deze functionoproep stoppen we in een variabele *naam.*

De tweede function, *userInputFunctions.toonBoodschap,* wijzigt de DOM maar retourneert niks. Daarom gebruiken we de IJSRuntime-method *InvokeVoidAsync().*

Dit alles mag enkel uitgevoerd worden bij een eerste weergave van de page, dus wanneer de parameter *firstRender* true is. Stoppen we deze code niet in een if-block dan wordt de naam telkens opnieuw gevraagd wanneer we een nieuw kwadraat berekenen.

- Probeer uit. Start de app en kies het menuitem *JS Interop.*
- Tik jouw naam in en klik op *OK.* Jouw naam wordt weergegeven op de page.

We maken een variant op het kwadraat-voorbeeld. Deze keer gebruiken we een elementreference om te verwijzen naar het DOM-element en niet een id.

We schrijven eerst een generieke JavaScript-function die de inhoud van een bepaald element vervangt door een opgegeven tekst. De function heeft 2 parameters : het element en de tekst.

- Voeg onderstaande JavaScript-function toe binnen de scripttag in *App.razor* :

```
window.setElementText = (element, text) => element.innerText = text;
```

Deze function is zeer algemeen en herbruikbaar: ze aanvaardt een element en een stuk tekst en stopt de tekst in de innerText van het element.

- Pas de code in *JSInterop.razor* als volgt aan :

```
@page "/JSInterop"
@inject IJSRuntime JSRuntime;
<h1>JSInterop</h1>
…
<p class="mt-2 h3">
    <span class="badge bg-success p-3" @ref="spanElement"></span>
</p>

@code {
    public int Getal { get; set; }
    private ElementReference spanElement;
    private async Task BerekenKwadraat()
    {
        var kwadraat = await JSRuntime.InvokeAsync<int>("kwadraat", Getal);
        await JSRuntime.InvokeVoidAsync("setElementText", spanElement,
            $"Het kwadraat van {Getal} is {kwadraat}.");
    }
    …
}
```

In de layout geven we de span waarin het resultaat wordt weergegeven een @ref-attribuut zodat we ernaar kunnen verwijzen. De span is voorlopig leeg.

In de code-section voegen we een member *spanElement* toe. Dit is een ElementReference die verwijst naar de resultaat-span. In de method *BerekenKwadraat()* laten we de JavaScript-function *kwadraat* het kwadraat berekenen. Het resultaat stoppen we in een lokale variabele *kwadraat.* Vervolgens roepen we de JavaScript-function *setElementText* op die de resultaatspan rechtstreeks opvult.

- Probeer opnieuw uit.

Opmerking : laat je niet verleiden tot het gebruik van JavaScript-functions om resultaten in jouw layout te tonen. Wat je kán doen met Blazor-code doe je ook best met Blazor-code en niet via JavaScript.

12.2 Oefening

Bij de eerste weergave van de page *JSInterop* wordt de naam van de gebruiker gevraagd en nadien getoond. Zorg ervoor dat daarna de focus automatisch in het inputveld komt te staan. Schrijf hiervoor een function die automatisch wordt uitgevoerd na het tonen van de welkomstboodschap.

12.3 .NET methods oproepen vanuit JavaScript

De omgekeerde beweging, het oproepen van .NET methods vanuit JavaScript-code, is een stuk ingewikkelder. Om een method van een bepaalde .NET class op te roepen vanuit JavaScript moeten we een reference leggen naar de class en die reference meegeven in de functionoproep.

Je kan ook een static method oproepen. Hier volstaat het de naam van de method mee te geven en de naam van de assembly waarin de static method gedefinieerd is. De naam van deze method moet wel uniek zijn binnen de assembly.

12.3.1 Oproepen van static methods

We beginnen met het makkelijkste : het oproepen van een static .NET method vanuit JavaScript. Je gebruikt hiervoor ofwel de function *DotNet.invokeMethod()* ofwel *DotNet.invokeMethodAsync()*.

De static .NET-method die je wil oproepen moet public zijn en voorzien van een [JSInvokable]-attribuut.

Een voorbeeld : via een klik op een knop activeren we een JavaScript-function die data ophaalt uit een static .NET method en vervolgens die data toevoegt aan een html-table.

- Maak in een nieuwe folder *Data* een static class *Data* :

```
using Microsoft.JSInterop;
…
public static class Data
{
    private static List<Persoon> personen = new List<Persoon> {
        new Persoon { Id = 1, Voornaam = "Jesse", Familienaam = "James" },
        new Persoon { Id = 2, Voornaam = "Jane", Familienaam = "Calamity" }
    };

    [JSInvokable]
    public static List<Persoon> GetPersonen()
    {
        return personen;
    }
}
```

In de static class *Data* definiëren we een static List<Persoon>. De method *GetPersonen()* die we vanuit JavaScript gaan oproepen is static, public én voorzien van een attribuut [JSInvokable].

- In de component *JSInterop.razor* voeg je een html-table en een button toe :

```
<h3>Personenlijst</h3>
<table class="table">
    <thead>
        <tr>
            <th>Id</th>
            <th>Voornaam</th>
            <th>Familienaam</th>
        </tr>
    </thead>
    <tbody>
    </tbody>
</table>
<button type="button" class="btn btn-primary" onclick="returnPersonenlijstJs()">
    Call static .NET method
</button>
```

De button activeert een JavaScript-function *returnPersonenlijstJs().* Deze function wordt rechtstreeks opgeroepen via het onclick-event van de button.

- Definieer de function *returnPersonenLijstJS* in *App.razor* :

```
…
<script>
    …
    window.returnPersonenlijstJs = function () {
        DotNet.invokeMethodAsync('BlazorServer', 'GetPersonen')
            .then(data => {
```

```
                    const tbody = document.querySelector("tbody");
                    for (let teller = 0; teller < data.length; ++teller) {
                        const tr = tbody.insertRow();
                        const idTd = tr.insertCell();
                        idTd.innerText = data[teller].id;
                        const voornaamTd = tr.insertCell();
                        voornaamTd.innerText = data[teller].voornaam;
                        const famnaamTd = tr.insertCell();
                        famnaamTd.innerText = data[teller].familienaam;
                    }
                }
            );
        };
    </script>
```

De code `DotNet.invokeMethodAsync('BlazorServer', 'GetPersonen')` roept de static .NET method *GetPersonen()* op uit de assembly *BlazorServer*. Vervang *BlazorServer* eventueel door de naam van jouw project als het een andere naam heeft.

Dit is een asynchronous call waarvan het resultaat even op zich kan laten wachten. Om zeker te zijn dat we over het resultaat beschikken vooraleer we de resultaatdata toevoegen aan de DOM gebruiken we de callback-function beschreven binnen *.then()*.

In die callback-function overlopen we de resultaatarray *data* en voegen we de gegevens van alle personen toe aan de html-table.

- Probeer uit. Een klik op de knop *`Call static .NET method`* toont de personenlijst.

Het resultaat :

Personenlijst

Id	Voornaam	Familienaam
1	Jesse	James
2	Jane	Calamity

Call static .NET method

Tip : als je niet het gewenste resultaat krijgt, druk dan F12 in de browser en controleer op eventuele foutmeldingen in de console.

12.3.2 Oproepen van een method van een object-instance

We kunnen vanuit JavaScript ook een method oproepen van een gewone class die dus niet static is.

Beschouw onderstaande class *Groeter.* Deze class beschikt over een string-property *Naam* en een method *Begroet.* De method *Begroet* returnt een begroeting die de naam bevat.

Om de method *Begroet* te kunnen oproepen vanuit JavaScript voorzien we de method (net als in de static class) van een attribuut [JSInvokable].

- Voeg, bijvoorbeeld in de folder *Models,* een class toe en noem deze *Groeter.*

- Voorzie de class *Groeter* van onderstaande code :

```
using Microsoft.JSInterop;
…
public class Groeter
{
    public Groeter(string naam)
    {
        Naam = naam;
    }

    public string Naam { get; set; }

    [JSInvokable]
    public string Begroet() => $"Hallo, {Naam}!";
}
```

Om de method *Begroet* te kunnen oproepen vanuit JavaScript moeten we het volgende doen :

1. Een nieuwe instance maken van het object *Groeter*
2. Een reference maken naar deze instance
3. Een JavaScript-function schrijven waarin de .NET-method via deze reference wordt opgeroepen.
4. Deze JavaScript-function oproepen in onze component en de reference naar het .NET-object meegeven als parameter.

- Voor stap 1, 2 en 4 neem je onderstaande code over in de code-section van *JSInterop.razor* :

```
…
public string JouwNaam { get; set; }

private DotNetObjectReference<Groeter> objectReference;

private async Task Begroeting()
{
    var groeter = new Groeter(JouwNaam);
    objectReference = DotNetObjectReference.Create(groeter);
    await JSRuntime.InvokeAsync<string>("callGroeter", objectReference);
}

public void Dispose()
{
    objectReference?.Dispose();
}
```

In de asynchrone method *Begroeting()* maken we een nieuwe instance van de class *Groeter.* De *Naam*-property vullen we op met de inhoud van een nieuwe componentmember *JouwNaam*. Verderop zullen we de component voorzien van een inputveld waarmee je de membervariabele *JouwNaam* opvult.

Na het aanmaken van de nieuwe Groeter-instance leggen we een reference naar dit object. We doen dit via de method *Create* van de class *DotNetObjectReference*. Dit is een class uit de namespace *Microsoft.JSInterop.* Het resultaat van de Create-method is een reference naar een Groeter-instance. De reference bewaren we in een membervariabele van het type *DotNetObjectReference<Groeter>*.

Nu we over een reference naar het .NET-object *groeter* beschikken kunnen we een JavaScript-function *callGroeter* starten. Deze function zal de meegegeven objectreference gebruiken om de

.NET-method op te roepen. We roepen de JavaScript-function op met *InvokeAsync<string>* omdat het resultaat van de function een string is.

In de method *Dispose()* ruimen we de resources op die we gebruikt hebben in de component. In dit geval is dit de objectreference. Vergeet dit niet, anders zit je met een memory-leak.

We zijn klaar voor stap 3 : een JavaScript-function schrijven die de .NET-method oproept.

- Voeg in *App.razor* in de script-tag onderstaande code toe :

```
window.callGroeter = function (objectReference) {
    return objectReference.invokeMethodAsync('Begroet')
        .then(r => {
            const p = document.querySelector("#boodschap");
            p.innerText = r;
        }
        );
};
```

Deze JavaScript-function krijgt als parameter de reference mee die verwijst naar het .NET-object *groeter.* Dit object heeft een method *Begroet*. We roepen deze method op door de method *invokeMethodAsync* op te roepen van de objectreference.

Deze oproep levert een string op. Van zodra dit resultaat beschikbaar is wordt de callbackfunction uitgevoerd die is aangegeven binnen *.then()*. In die callbackfunction tonen we het resultaat (r) op onze page door de p-tag met id *boodschap* te zoeken en daarvan de innerText op te vullen met de resultaattekst.

De code is klaar, we moeten nu enkel nog de UI toevoegen in onze component *JSInterop.razor*.

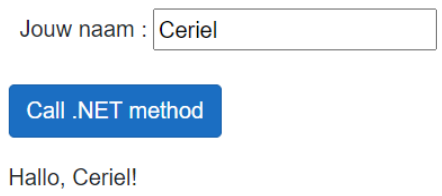
- Voeg onderstaande layout-code toe in *JSInterop.razor* :

```
<div class="my-4 ms-2">
    <span for="jouwnaam">Jouw naam : </span>
    <input id="jouwnaam" @bind-value="JouwNaam" />
</div>
<button type="button" class="btn btn-primary" @onclick="Begroeting">
    Call .NET method
</button>
<p id="boodschap" class="mt-3"/>
```

We voorzien een inputveld dat gebonden is aan de string-property *JouwNaam*. Een klik op de button start de method *Begroeting* die je op de vorige pagina hebt gemaakt en waarin stappen 1, 2 en 4 worden uitgevoerd. Het eindresultaat wordt getoond in de paragraph met id *boodschap.*

- Probeer uit.

Het resultaat :



12.4 JavaScript isolation

Alle JavaScript-code wordt nu geladen bij het opstarten van de applicatie. We kunnen het laden van JavaScript-code die hoort bij een bepaalde component uitstellen tot wanneer de component wordt gebruikt. We noemen dit JavaScript isolation.

We coderen de JavaScript-code in aparte .js-bestanden die we onderbrengen in de folder *wwwroot* of een subfolder ervan. Vervolgens laden we die bestanden in de method *OnAfterRenderAsync().*

We proberen dit uit.

- Voeg in de folder *wwwroot* een subfolder *js* toe.
- Maak in de folder *js* een nieuwe .js-file en noem deze *scripts.js*.
- Voeg onderstaande code toe aan het bestand *scripts.js* :

```
export function NaamInvoeren(boodschap) {
    return prompt(boodschap, 'Tik hier jouw naam...');
}

export function BoodschapTonen(boodschap) {
    return alert(boodschap);
}
```

Het bestand *scripts.js* bevat twee functions. De function *NaamInvoeren* toont een dialoogvenster waarin de gebruiker een naam kan invoeren. De parameter boodschap is de tekst die boven het invulveld verschijnt. Dit invulveld toont aanvankelijk de tekst *Tik hier jouw naam…*

De tweede function toont een tekst in een dialoogvenster. De gebruiker kan het venster sluiten door op een OK-knop te klikken.

- Voeg nu in de folder Pages een nieuwe component toe en noem deze *JSIsolation.razor.*
- Voeg er onderstaande code aan toe :

```
@page "/jsisolation"
@inject IJSRuntime JSRuntime
<h3>JSIsolation</h3>
<button @onclick="VraagNaam" class="btn btn-primary">Vraag naam</button>
<button @onclick="ToonNaam" class="btn btn-primary">Toon de naam</button>
@code {

    string naam = string.Empty;

    IJSObjectReference module;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JSRuntime.InvokeAsync<IJSObjectReference>(
                "import", "./js/scripts.js");
        }
    }

    async Task VraagNaam()
    {
        naam = await module.InvokeAsync<string>("NaamInvoeren",
            "Vul jouw naam in : ");
    }
```

```
        async Task ToonNaam()
        {
            await module.InvokeVoidAsync("BoodschapTonen", $"Jouw naam is {naam}");
        }
    }
```

In de lifecycle method *OnAfterRenderAsync* wordt er een IJSObjectReference gelegd naar het scriptbestand *scripts.js* en bewaard in een member variabele *module.*

Op de page vinden we twee buttons. De eerste button activeert de method *VraagNaam*. In deze method roepen we via de IJSObjectReference *module* de JavaScript-function *NaamInvoeren* op. Als parameter geven we de tekst mee die in het invulveld moet weergegeven worden. Het resultaat van deze function is een string en deze wordt opgeslagen in de membervariabele *naam.*

De tweede button activeert de method *ToonNaam.* Daar roepen we de JavaScript-function *BoodschapTonen* op. Naast de naam van de op te roepen function geven we ook de af te beelden boodschap mee als parameter. Deze function heeft geen resultaat dus gebruiken we hier de method *InvokeVoidAsync* i.p.v. *InvokeAsync.*

- Probeer uit : start de app en druk op F12.
- Bekijk het tabblad *Network.* Doe een refresh van de pagina zodat je ziet wat er allemaal geladen wordt. Het bestand *scripts.js* wordt (nog) niet geladen.
- Navigeer naar *./jsisolation*.
- Bekijk opnieuw het tabblad *Network.* Het bestand *scripts.js* werd nu wel geladen. De scripts kunnen gebruikt worden.

13 Security

Net als in een ASP.NET Core webapplicatie kunnen we in een Blazor app authentication en authorization gebruiken om de app te beveiligen.

De manier waarop we authentication en authorization toepassen binnen een **Blazor** app lijkt heel erg op de werkwijze die we in een ASP.NET Core webapplicatie gebruikten. We gaan ook hier gebruik maken van Identity Server. Daarvoor hebben we een backend nodig waarop we de nodige security-checks kunnen uitvoeren. In een Blazor Server app hebben we die backend reeds. Bij een Blazor WebAssembly app ligt dit enigszins anders. Die draait op de client. We kunnen echter bij de creatie van een Blazor WebAssembly app een backend toevoegen.

Vooraleer we effectief aan de slag gaan in een Blazor Server of WebAssembly app overlopen we even enkele belangrijke Blazorcomponenten die we kunnen gebruiken om de security toe te passen.

13.1 AuthorizeView component

In een Blazor app gebruik je een AuthorizeView-component om login-afhankelijke content te laten zien. Je kan UI-onderdelen laten zien wanneer een user zich heeft ingelogd of tot een bepaalde role behoort.

De templated component AuthorizeView beschikt over een context-variabele van het type AuthenticationState, waarmee je informatie over de ingelogde user kan verkrijgen. Een template *Authorized* beschrijft de UI voor een ingelogde gebruiker, de template *NotAuthorized* de UI voor een niet-ingelogde gebruiker.

Voorbeeld :

```
<AuthorizeView>
    <Authorized>
        <h1>Hallo, @context.User.Identity.Name!</h1>
        <p>Je kan deze content enkel zien wanneer je ingelogd bent.</p>
    </Authorized>
    <NotAuthorized>
        <h1>Authenticatiefout</h1>
        <p>Je bent niet ingelogd.</p>
    </NotAuthorized>
</AuthorizeView>
```

Je vindt een voorbeeld van het gebruik van de component AuthorizeView in NavMenu.razor.

Een AuthorizeView kan de attributen Roles of Policy hebben. Afhankelijk van de waarde van die attributen kan een afwijkende UI getoond worden :

Role-based authorization :

```
<AuthorizeView Roles="admin, superuser">
    <p>You can only see this if you're an admin or superuser.</p>
</AuthorizeView>
```

Policy-based authorization :

```
<AuthorizeView Policy="content-editor">
    <p>You can only see this if you satisfy the "content-editor" policy.</p>
</AuthorizeView>
```

13.2 [Authorize] attribuut

Je geeft aan dat een page enkel mag weergegeven worden voor ingelogde gebruikers door een [Authorize] attribuut op te nemen bovenaan de razor component :

```
@attribute [Authorize]
```

Het attribuut geldt voor de ganse page. Wil je afhankelijk van de login-status van de gebruiker een deel van een page al dan niet laten weergeven, dan gebruik je een AuthorizeView-component.

Je kan het Authorize-attribuut ook voorzien van een parameter Roles of Policy :

```
@attribute [Authorize(Roles = "admin, superuser")]
```

```
@attribute [Authorize(Policy = "content-editor")]
```

Als er geen role of policy is vermeld dan krijgen alle ingelogde users toegang tot de page.

De component Auth.razor bevat een attribuut [Authorize] en is dus enkel bruikbaar voor ingelogde gebruikers.

13.3 AuthorizeRouteView

In *Routes.razor* geeft de Router-component in de <Found>-template aan welke content er moet weergegeven worden wanneer de route wordt gevonden. We kunnen deze template nog verder verfijnen door gebruik te maken van een AuthorizeRouteView-component. Deze bevat de templates NotAuthorized en Authorizing. De NotAuthorized-template bevat de content die moet weergegeven worden wanneer de gebruiker de gevraagde page niet mag zien. De Authorizing-template geeft content weer die getoond wordt in afwachting van authorisatie.

Voorbeeld :

```
<Router AppAssembly="typeof(Program).Assembly">
    <Found Context="routeData">
        <AuthorizeRouteView RouteData="routeData"
                            DefaultLayout="typeof(Layout.MainLayout)">
            <NotAuthorized>
                <h1>Geen toegang</h1>
                <p>Je bent niet gemachtigd om deze pagina te bekijken.</p>
            </NotAuthorized>
            <Authorizing>
                <h1>Even geduld</h1>
                <p>Het authenticatieprocess is momenteel bezig</p>
            </Authorizing>
        </AuthorizeRouteView>
        <FocusOnNavigate RouteData="@routeData" Selector="h1" />
    </Found>
</Router>
```

13.4 Security in een Blazor WebAssembly app

We passen nu security toe in een Blazor app. We beginnen met een WebAssembly app.

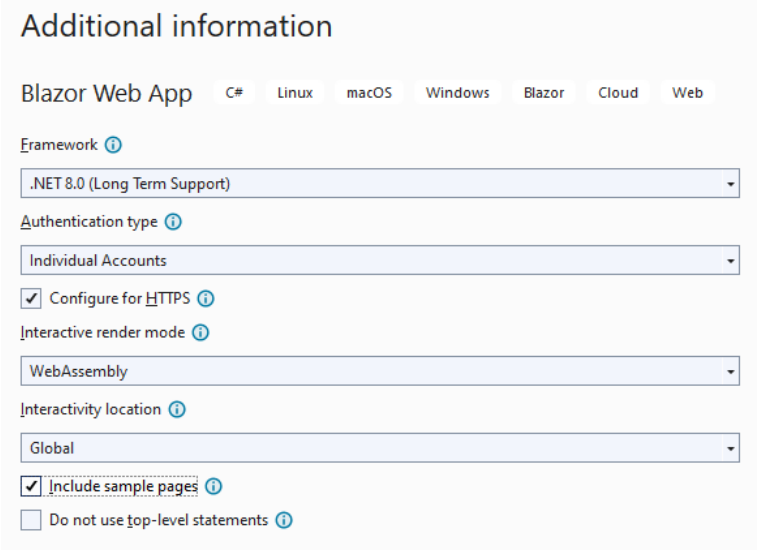
Er zijn verschillende manieren om authorization technisch mogelijk te maken. Wij gaan identity server gebruiken omdat we dit ook in ASP.NET Core hebben gedaan en deze techniek zowel in Blazor WebAssembly als Blazor Server kan gebruikt worden.

Wanneer een gebruiker een page opvraagt die voorzien is van een [Authorize]-attribuut dan wordt hij meestal naar een login-pagina gestuurd met URL /authentication/login. In die loginpage zal je geredirect worden naar een zogenaamde authorization endpoint. Deze endpoint wordt gehost buiten de WebAssembly app en controleert eerst of de gebruiker al dan niet gekend is. Is dit niet het geval dan kan de gebruiker zich registreren. Wordt de gebruiker wél gevonden dan wordt hij naar een login callback endpoint geredirect. Aan de hand van de opgezochte authenticationgegevens wordt daar beslist of de gebruiker geauthorizeerd is om de gevraagde page te bekijken.

13.4.1 Een beveiligde Blazor app

We proberen dit uit in een nieuwe Blazor WebAssembly app met security.

- Voeg aan de solution een nieuw Blazor Web App project toe en noem deze *WebAssemblySecurity.* Klik op *Next.*
- Kies bij *Authentication type* voor *Individual Accounts*.
- Kies bij *Interactive render mode* voor *WebAssembly.* De *Interactivity location* houden we *Global.*
- Klik op *Create.*



Jouw Blazor WebAssembly app bestaat nu uit twee projecten : *WebAssemblySecurity en WebAssemblySecurity.Client.*

In het project *WebAssemblySecurity* vind je in de folder *Components/Account/Pages* reeds heel wat Blazor componenten om in te loggen, je te registreren en nog veel meer pages die enige interactie met de securitydatabase vereisen.

In Program.cs zijn de nodige securityservices toegevoegd aan de middleware pipeline. In de folder Data vind je een DbContext-class en een migration om de securitydatabase aan te maken.

In het tweede project, *WebAssemblySecurity.Client* komen vooral pages die meer met de eigenlijke inhoud van de app hebben te maken en niet met de security van de app.

In dit project is een extra dependency naar de package *Microsoft.AspNetCore.Components. WebAssembly.Authentication* toegevoegd. In *Routes.razor* wordt een AuthorizeRouteView gebruikt i.p.v. een RouteView.

De component *NavMenu.razor* bevat een menu met een aantal menuitems. De menuitems *Home*, *Counter* en *Weather* kennen we reeds uit een niet-beveiligde Blazor app. De menuoptie *Auth Required* leidt tot een beveiligde page *Auth.* Wanneer je de source van deze component bekijkt zie je dat er bovenaan een attribuut [Authorize] opgenomen is. Is de gebruiker ingelogd dan zijn er nog twee extra menuitems zichtbaar. Met de eerste kan de gebruiker zijn account beheren, met de tweede kan er uitgelogd worden. Wanneer de gebruiker niet ingelogd is dan bieden de twee laatste menuopties de mogelijkheid om zich te registreren en in te loggen.

13.4.2 De security database

Bij de aanmaak van de Blazor WebAssembly app werd in het opstartproject in de folder *Data* een modelclass *ApplicationUser* aangemaakt. Aan deze class, die erft van IdentityUser, kunnen eventueel extra property's toegevoegd worden. In dezelfde folder vind je ook de dbcontext-class *Application-DbContext*.

In de folder *Data/Migrations* vinden we reeds een eerste migration. Door die door te voeren maak je de securitydatabase aan.

De naam van de securitydatabase wordt bepaald door de connectionstring in *appsettings.json.* Je vindt dit bestand in de root van het project WebAssemblySecurity. De naam van de database begint standaard met 'aspnet-', dan de naam van het project en vervolgens een unieke code. Wanneer je veel WebAssembly Blazor apps maakt met security wordt het soms moeilijk om te weten welke database er bij welke app hoort. Je kan daarom de securitydatabase gerust een zinvollere naam geven door de connectionstring te wijzigen.

- Wijzig eventueel de naam van de database in *appsettings.json.*
- Open de Package Manager Console en positioneer je in het project *WebAssemblySecurity*.
- Voer de migration door met het commando **dotnet ef database update**

13.4.3 De beveiligde app uitproberen

Vooraleer we de applicatie opstarten wijzigen we nog één instelling. Standaard moet een gebruiker zijn registratie bevestigen via mail. Deze instelling kan je uitschakelen in *Program.cs* door *RequireConfirmedAccount* = ***true*** te wijzigen in *RequireConfirmedAccount* = ***false****.*

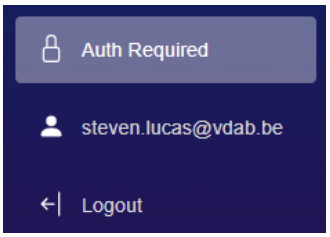
- Wijzig de code in *Program.cs* als volgt :

```
builder.Services.AddIdentityCore<ApplicationUser>(options =>
            options.SignIn.RequireConfirmedAccount = false)
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddSignInManager()
        .AddDefaultTokenProviders();
```

We testen de database nu door ons te registreren in de app.

- Start de applicatie.
- Kies het menuitem *Counter*. Dit is niet beveiligd. Je kan de component gewoon gebruiken. Hetzelfde geldt voor het menuitem *Weather.*
- Kies het menuitem *Auth Required*. Zoals de naam van het menuitem zelft zegt is dit een beveiligde pagina. Om die te bekijken moet je ingelogd zijn.
- Je wordt verzocht om in te loggen. Kies onderaan de link *Register as a new user*.
- Registreer je. Vergeet daarbij niet een sterk paswoord te gebruiken.

Na een succesvolle registratie ben je ook ingelogd en krijg je meteen de component *Auth Required* te zien. De menuitems Login en Register zijn vervangen door een menuitem met jouw loginnaam en een menuitem Logout.



13.5 Security in een Blazor Server app

Ook een Blazor Server app kan je beveiligen. We proberen het uit.

- Voeg een nieuw Blazor Web App project toe aan de solution en noem deze BlazorServerSecurity. Klik op *Next*.
- Kies bij *Authentication type* voor *Individual Accounts,* zet de *Interactive render mode* op *Server.* De overige instellingen laat je zoals ze ingesteld zijn. Klik op *Create.*

Na de creatie van jouw Blazor Server app vind je in de folder *Data/Migrations* een initiële migration waarmee je de security-database kan aanmaken. We voeren de migration nu door.

- Wijzig eventueel de naam van de securitydatabase in *appsettings.json.*
- Open de Package Manager Console.
- Positioneer je in de projectfolder.
- Voer de migration door via het commando **dotnet ef database update**.

Als je dat wil kan je ook in deze app in Program.cs de signin-option RequireConfirmedAccount op false zetten. Wanneer je je registreert hoeft jouw login dan niet meer bevestigd te worden.

Wanneer je de app opstart zie je dat de app dezelfde menuitems bevat als een beveiligde WebAssembly app. De twee onderste menuitems veranderen na een geldige loginactie ook hier in een link naar een profilepage en een logout-optie.

We passen nu de security toe. We doen dit voor de component Weather.

- Open *Weather.razor* en voeg de vetgedrukte code toe :

```
@page "/weather"
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]
<PageTitle>Weather</PageTitle>

<h1>Weather</h1>
…
```

- Start de app en kies het menuitem *Weather*. Je wordt gevraagd je aan te melden.
- Registreer je en bevestig eventueel jouw account. Deze keer kan je de component *Weather* wel bekijken.

We gaan de security in deze app nog verder verfijnen door enkel admins toegang te geven tot de component Weather.

We voegen net als bij de WebAssembly app eerst extra gegevens toe aan de securitydatabase. Daarvoor gebruiken we opnieuw het security-script.

- Start SSMS en open het bestand *Security.sql.*
- Wijzig op de eerste lijn van dit script de naam van de securitydatabase naar de database-naam die je vindt in appsettings.json.
- Voer het script uit.

We moeten in Program.cs nog code toevoegen om roles te kunnen gebruiken.

- Wijzig de code in *Program.cs* als volgt :

```
…
builder.Services.AddIdentityCore<ApplicationUser>(options =>
        options.SignIn.RequireConfirmedAccount = false)
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddSignInManager()
    .AddDefaultTokenProviders();
…
```

We kunnen nu het Authorize-attribuut in de component *Weather.razor* aanpassen zodat enkel Admins toegang krijgen tot deze component.

- Wijzig het Authorize-attribuut in *FetchData.razor* als volgt :

```
@attribute [Authorize(Roles = "Admin")]
```

- Je kan uitproberen : log in als gewone user en kies het menuitem *Weather.* Je krijgt geen toegang. Log daarna in als admin, je ziet de weersvoorspellingen.

14 Een Web API aanspreken vanuit Blazor

In een Blazor app zullen we ook data willen ophalen uit een database. In een Blazor Server app kunnen we op dezelfde manier tewerk gaan als in de ASP.NET Core MVC cursus : je voegt class library's toe voor de datalaag die de database aanspreekt en voor de servicelaag die de business logic bevat.

Bij een Blazor WebAssembly app moeten we op een andere manier tewerk gaan want de ganse app wordt op de client uitgevoerd. Hier zullen we een Web API aanspreken om de data op te halen.

Als voorbeeld maken we een Blazor WebAssembly app waarmee we de gegevens van onze contacten bijhouden.

14.1 Opzetten van de applicatiestructuur

We zullen dus de gegevens van onze contacten presenteren in een Blazor WebAssembly app. Voor de backend gebruiken we een WebAPI-project. Beide projecten zullen gebruik maken van één of meerdere modelclasses. Die stoppen we in een aparte class library.

- Maak in een nieuwe solution *Contacten* een Blazor Web App *Contacten.Web*. Klik op *Next.*
- Kies bij *Interactive render mode* de optie *WebAssembly.*
- Behoud de standaardinstellingen. Dus geen authentication en de interactivity location is global. Kies *Create*.
- Voeg aan de solution een .NET Class Library toe en noem deze *Contacten.Models*.
- Voeg ook een *ASP.NET Core Web API* toe en noem deze *Contacten.Data.* Behoud de standaardopties (mét controllers) en klik op *Create*.

14.2 Models

Ons datamodel bestaat uit slechts één class, de class *Contact*.

- Verwijder in het project *Contacten.Models* de class *Class1.cs* en voeg een nieuwe class *Contact.cs* toe :

```
public class Contact
{
    public int ContactId { get; set; }
    public string Voornaam { get; set; }
    public string Naam { get; set; }
    public string Adres { get; set; }
    public string Postcode { get; set; }
    public string Gemeente { get; set; }
    public string Telefoon { get; set; }
    public string Email { get; set; }
    public DateTime Geboortedatum { get; set; }
}
```

Deze class zullen we in de Blazor app én in de WebAPI willen gebruiken. We voegen de nodige references toe.

- Klik in de Solution Explorer met de rechtermuistoets op het project *Contacten.Data* en kies *Add – Project Reference…*
- Zet een vinkje voor *Contacten.Models* en klik op OK.

- Voeg op dezelfde manier ook aan het project *Contacten.Web.Client* een reference toe naar *Contacten.Models.*

14.3 Web API

In het Web API project, *Contacten.Data*, vind je een class *WeatherForecast.cs* en een WeatherForecast-controller. Je mag beide classes verwijderen.

14.3.1 Opbouw van de database Contacten

Als eerste stap bouwen we een nieuwe database *Contacten* op. We beginnen met het toevoegen van een connectionstring in het appsettings.json-bestand.

- Voeg in *appsettings.json* een connectionstring toe :

```
…
"ConnectionStrings": {
  "ContactenConnection":
    "server=(localdb)\\mssqllocaldb;Database=Contacten;Trusted_Connection=True;"
}
}
```

We gebruiken uiteraard EntityFrameworkCore dus voeg je de nodige NuGet packages toe.

- Installeer de packages *Microsoft.EntityFrameworkCore.SqlServer* en *Microsoft.EntityFrameworkCore.Design.*

Je maakt nu de DbContext-class in een nieuwe folder *Models.*

- Voeg in het API-project een folder *Models* toe.
- Voeg in die folder een class *AppDbContext.cs* toe en voorzie deze van onderstaande code :

```
using Contacten.Models;
using Microsoft.EntityFrameworkCore;
…
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
    {
    }

    public DbSet<Contact> Contacten { get; set; }
}
```

Om de database te voorzien van wat testdata override je de method OnModelCreating(). Dit kan snel door te tikken *override onm* en op Tab te drukken :

```
override onm
          OnModelCreating(ModelBuilder modelBuilder)
```

- Vul de method *OnModelCreating* aan met volgende code :

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

    modelBuilder.Entity<Contact>().HasData(
        new Contact { ContactId = 1, Voornaam = "Jesse", Naam = "James",
```

```
                Adres = "Somersstraat 22", Postcode = "2018",
                Gemeente = "Antwerpen", Telefoon = "032021711",
                Email = "jesse.james@vdab.be",
                Geboortedatum = new DateTime(1966, 1, 1) });

        modelBuilder.Entity<Contact>().HasData(
            new Contact { ContactId = 2, Voornaam = "Jane", Naam = "Calamity",
                Adres = "Interleuvenlaan 2", Postcode = "3001",
                Gemeente = "Heverlee", Telefoon = "016380000",
                Email = "jane.calamity@vdab.be",
                Geboortedatum = new DateTime(1967, 2, 2) });

        modelBuilder.Entity<Contact>().HasData(
            new Contact { ContactId = 3, Voornaam = "Billy", Naam = "The Kid",
                Adres = "Vlamingstraat 10", Postcode = "8560",
                Gemeente = "Wevelgem", Telefoon = "056438080",
                Email = "billy.thekid@vdab.be",
                Geboortedatum = new DateTime(1968, 3, 3) });

        modelBuilder.Entity<Contact>().HasData(
            new Contact { ContactId = 4, Voornaam = "Sarah", Naam = "Bernhardt",
                Adres = "Industrieweg 50", Postcode = "9032",
                Gemeente = "Wondelgem", Telefoon = "092541111",
                Email = "sarah.bernhardt@vdab.be",
                Geboortedatum = new DateTime(1969, 4, 4)});
    }
```

We configureren nu de DbContext-class in *Program.cs*.

- Open in het API-project *Program.cs* en voeg onderstaande vetgedrukte code toe :

```
using Contacten.Data.Models;
using Microsoft.EntityFrameworkCore;
…
builder.Services.AddControllers();
builder.Services.AddDbContext<AppDbContext>(options => options.UseSqlServer(
    builder.Configuration.GetConnectionString("ContactenConnection")));
…
```

We kunnen nu een initiële migration toevoegen en die ook uitvoeren.

- Open de package manager console en positioneer je in de folder *Contacten.Data*.
- Tik de commando's *dotnet ef migrations add InitialMigration* en daarna *dotnet ef database update.*

14.3.2 De Contacten-repository

We maken nu een repository waarmee we de CRUD-operaties kunnen uitvoeren op de contactentabel.

We maken eerst de interface *IContactenRepository.cs.*

- Voeg in het API-project in de folder *Models* onderstaande interface toe :

```
public interface IContactenRepository
{
    Task<IEnumerable<Contact>> GetContacten();
    Task<Contact> GetContact(int contactId);
    Task<Contact> AddContact(Contact contact);
    Task<Contact> UpdateContact(Contact contact);
    Task<Contact> DeleteContact(int contactId);
}
```

Deze interface beschrijft een contactenrepository die we zo meteen zullen implementeren in een SQLServer repository. In de interface vind je de CRUD-operaties terug. Deze operaties zijn beschreven als asynchrone methods.

- Voeg in de folder *Models* een class *ContactenRepository.cs* toe :

```
using Contacten.Models;
using Microsoft.EntityFrameworkCore;
…
public class ContactenRepository : IContactenRepository
{
    private readonly AppDbContext context;

    public ContactenRepository(AppDbContext context)
    {
        this.context = context;
    }

    public async Task<Contact> AddContact(Contact contact)
    {
        var result = await context.Contacten.AddAsync(contact);
        await context.SaveChangesAsync();
        return result.Entity;
    }

    public async Task<Contact> DeleteContact(int contactId)
    {
        var result = await context.Contacten
            .FirstOrDefaultAsync(c => c.ContactId == contactId);
        if (result != null)
        {
            context.Contacten.Remove(result);
            await context.SaveChangesAsync();
            return result;
        }
        return null;
    }

    public async Task<Contact> GetContact(int contactId)
    {
        return await context.Contacten
            .FirstOrDefaultAsync(c => c.ContactId == contactId);
    }

    public async Task<IEnumerable<Contact>> GetContacten()
    {
        return await context.Contacten.ToListAsync();
    }

    public async Task<Contact> UpdateContact(Contact contact)
    {
        var result = await context.Contacten
            .FirstOrDefaultAsync(c => c.ContactId == contact.ContactId);
        if (result != null)
        {
            result.Voornaam = contact.Voornaam;
            result.Naam = contact.Naam;
            result.Adres = contact.Adres;
            result.Postcode = contact.Postcode;
            result.Gemeente = contact.Gemeente;
            result.Telefoon = contact.Telefoon;
```

```
                result.Email = contact.Email;
                result.Geboortedatum = contact.Geboortedatum;
                await context.SaveChangesAsync();
                return result;
            }
            return null;
        }
    }
```

Deze class implementeert de interface *IContactenRepository*. Dit betekent dat we de 5 methods uit de interface beschrijven. In die methods maken we gebruik van een dbcontext. Die injecteren we via de constructor en slaan die op in een private variabele.

In de method AddContact voegen we de meegegeven contact-parameter toe aan de Contacten-dbset. We bewaren de wijzigingen en retourneren het toegevoegde contact. Op deze manier beschikken we over de waarde van de id van het nieuwe contact, die via autonummering is toegekend.

Om een contact te kunnen deleten zoeken we eerst het contact op via de id. Indien dit een resultaat oplevert dan verwijderen we het contact, bewaren we de wijzigingen en retourneren we het verwijderde contact. Wordt het contact niet gevonden dan retourneren we null.

In de method GetContact() zoeken we een contact op a.d.h.v. de id. In GetContacten() retourneren we de ganse dbset, omgezet in een List<Contact>.

Om een contact te updaten zoeken we het contact eerst op. Indien dit iets oplevert passen we de velden aan, bewaren we de wijzigingen en retourneren we het gewijzigde contact. Wordt het contact niet gevonden dan retourneren we null.

14.3.3 De API-controller

De repository is klaar en kan nu gebruikt worden in een nieuwe API-controller.

- Voeg in de folder *Controllers* een nieuwe, empty **API** controller toe en noem deze *ContactenController*.

In deze controller injecteren we de repositoryservice via de constructor en slaan die op in een private variabele.

- Vul de code in *ContactenController.cs* als volgt aan :

```
[Route("api/[controller]")]
[ApiController]
public class ContactenController : ControllerBase
{
    private readonly IContactenRepository contactenRepository;

    public ContactenController(IContactenRepository contactenRepository)
    {
        this.contactenRepository = contactenRepository;
    }
}
```

Het injecteren van de repository kan enkel als deze geconfigureerd is in *Program.cs*. We doen dit nu.

- Voeg in *Program.cs* onderstaande vetgedrukte code toe :

```
…
builder.Services.AddScoped<IContactenRepository, ContactenRepository>();
…
```

We vullen nu de verschillende CRUD-operaties aan in onze API. We starten met het ophalen van alle contacten.

- Voeg onderstaande actionmethod toe in ContactenController.cs :

```
[HttpGet]
public async Task<ActionResult> GetContacten()
{
    try {
        return Ok(await contactenRepository.GetContacten());
    }
    catch (Exception) {
        return StatusCode(StatusCodes.Status500InternalServerError,
            "Database error");
    }
}
```

Voor het ophalen van de contacten gebruiken we uiteraard een HttpGet. Levert het ophalen van de contacten een goed resultaat op dan is het resultaat van deze method Ok. Zoniet dan retourneren we een statuscode 500.

Op een gelijkaardige manier kunnen we ook één contact opzoeken.

- Voeg onderstaande actionmethod toe:

```
using Contacten.Models;
…
[HttpGet("{id:int}")]
public async Task<ActionResult<Contact>> GetContact(int id)
{
    try {
        var result = await contactenRepository.GetContact(id);
        if (result == null) return NotFound();
        return result;
    }
    catch (Exception) {
        return StatusCode(StatusCodes.Status500InternalServerError,
            "Database error");
    }
}
```

We gebruiken opnieuw een Get maar deze keer met een int-parameter *id*. We zoeken het contact op in de repository en retourneren het resultaat. Wordt er niks gevonden dan retourneren we een NotFound(). Levert de methodoproep een fout op dan geven we een statuscode 500 terug.

Voor het toevoegen van een nieuw contact gebruiken we een Post action method.

- Voeg onderstaande actionmethod toe :

```
[HttpPost]
public async Task<ActionResult<Contact>> CreateContact(Contact contact)
{
    try {
        var createdContact = await contactenRepository.AddContact(contact);
        return CreatedAtAction(nameof(CreateContact),
```

```
                new { id = createdContact.ContactId }, createdContact);
        }
        catch (Exception) {
            return StatusCode(StatusCodes.Status500InternalServerError,
                "Database error");
        }
    }
```

Na het succesvol toevoegen van het nieuwe contact retourneren we met de CreatedAtAction-method de URI waar het nieuwe contact kan opgevraagd worden. Ook de id van het nieuwe contact en het nieuwe contact wordt geretourneerd. Doet er zich een fout voor dan retourneren we een statuscode 500.

- Voor de update van een contact voeg je onderstaande actionmethod toe :

```
[HttpPut("{id:int}")]
public async Task<ActionResult<Contact>> UpdateContact(int id, Contact contact)
{
    try {
        var contactToUpdate = await contactenRepository.GetContact(id);

        if (contactToUpdate == null)
            return NotFound($"Contact with id = {id} is not found");

        return await contactenRepository.UpdateContact(contact);
    }
    catch (Exception) {
        return StatusCode(StatusCodes.Status500InternalServerError,
           "Error updating data");
    }
}
```

Om het contact te updaten zoeken we eerst het contact op. Als die niet gevonden wordt retourneren we een NotFound.

Rest ons nog het verwijderen van een contact.

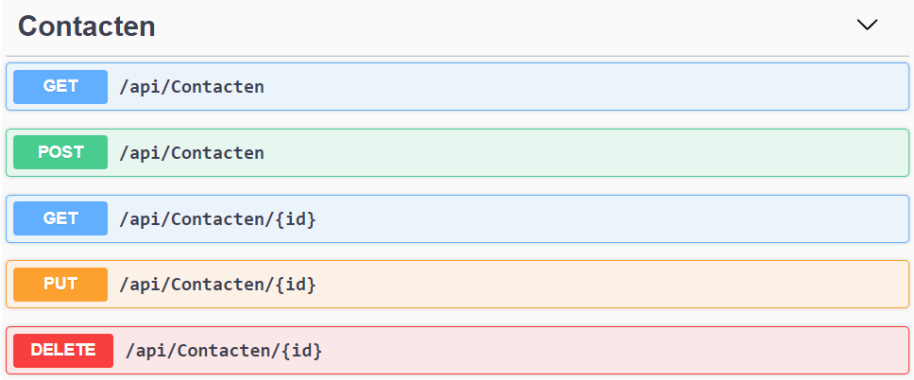
- Voeg hiervoor onderstaande actionmethod toe :

```
[HttpDelete("{id:int}")]
public async Task<ActionResult<Contact>> DeleteContact(int id)
{
    try {
        var contactToDelete = await contactenRepository.GetContact(id);

        if (contactToDelete == null)
            return NotFound($"Contact with id = {id} is not found");

        return await contactenRepository.DeleteContact(id);
    }
    catch (Exception) {
        return StatusCode(StatusCodes.Status500InternalServerError,
            "Error deleting data");
    }
}
```

Deze method bevat gelijkaardige code als bij de update : het contact wordt gezocht. Als het niet wordt gevonden of er doet zich een andere fout voor dan retourneren we een fout. Wordt het contact wel gevonden dan verwijderen we het contact.

Je kan de werking van de API uittesten met Swagger.

- Maak van het project *Contacten.Data* het opstartproject.
- Start de applicatie, je krijgt nu de Swagger indexpage te zien met knoppen voor de diverse API-methods.



- Klik op de bovenste GET-button.
- Deze method heeft geen parameters dus klik je meteen op Try it out en vervolgens op de knop Execute. Als alles goed gaat zie je onderaan de code 200 en alle contacten in JSON-formaat.
- Je kan eventueel ook de andere methods uitproberen. Zorg er wel voor dat je een geldige set contacten overhoudt.

14.4 Presentatie in Blazor WebAssembly

We gaan de API nu aanspreken vanuit de Blazor WebAssembly app. Het aanleveren van de API-data is een service die we gaan injecteren in verschillende componenten. We beginnen met het weergeven van alle contacten in een component *ContactenLijst*.

14.4.1 ContactenLijst

We bouwen eerst de service op die de API zal aanspreken.

- Voeg in het project *Contacten.Web.Client* een nieuwe folder *Services* toe.
- Voeg in de folder *Services* een nieuwe class *ContactenService.cs* toe.
- Voeg onderstaande code toe in *ContactenService.cs* :

```
using Contacten.Models;
using System.Text.Json;
…
public class ContactenService
{
    private readonly HttpClient httpClient;
    public ContactenService(HttpClient httpClient) {
        this.httpClient = httpClient;
    }

    public async Task<IEnumerable<Contact>> GetContacten()
    {
        var contacten = await httpClient.GetStringAsync("api/contacten");
        var options = new JsonSerializerOptions
        {
            PropertyNameCaseInsensitive = true
        };
```

```
            return JsonSerializer.Deserialize<IEnumerable<Contact>>(
                contacten, options);
        }
    }
```

In de serviceclass injecteren we een HttpClient. In de method *GetContacten* gebruiken we de method *GetStringAsync* van deze class om een lijst van contacten op te vragen. Het resultaat van deze method is een string die de contacten voorstelt in json-formaat. Die deserializen we naar een IEnumerable<Contact>.

> In bovenstaande code zetten we een option PropertyNameCaseInsensitive op true en geven die mee bij het deserializen. JsonSerializer deserializet immers standaard case-sensitive. Dit is een probleem aangezien de property's in de jsonstring met een kleine letter beginnen en de namen van de property's in de class Contact met een hoofdletter. Zonder deze extra option zou de deserialisatie mislukken. Dit probleem doet zich niet voor in ASP.NET Core, wél in andere omstandigheden zoals een Blazor WebAssembly app.

We configureren nu de contactenservice en de httpclient in *Program.cs* in Contacten.Web.Client.

- Wijzig de code als volgt :

```
using Contacten.Web.Client.Services;
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;

var builder = WebAssemblyHostBuilder.CreateDefault(args);

builder.Services.AddTransient<ContactenService>();
builder.Services.AddSingleton(
    new HttpClient { BaseAddress = new Uri("https://localhost:7266/") });

await builder.Build().RunAsync();
…
```

We voegen hier twee services toe : de ContactenService en de HttpClient. Voor deze laatste moeten we het BaseAddress opgeven. Je vindt het baseaddress van jouw WebAPI in de projectproperties op het tabblad *Debug* bij *General.* Wanneer je klikt op *Open debug launch profiles* dan zie je bij *App URL* het baseaddress. Neem het baseaddress voor https.

- Vervang in bovenstaande code het baseaddress door het baseaddress van jouw WebAPI.

We kunnen nu een nieuwe component maken om de contactenlijst weer te geven.

- Verwijder in jouw Blazor app de componenten *Counter.razor* en *Weather.razor.*
- Verwijder in *NavMenu.razor* de menuitems die naar deze componenten verwijzen en vervang door één nieuw menuitem :

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="contactenlijst">
        <span class="bi bi-person" aria-hidden="true"></span> Contacten
    </NavLink>
</div>
```

- Voeg in *App.razor* een verwijzing toe naar de css-file voor de bootstrap-icons. Hoe je dit doet lees je op p. 22.
- Wijzig ook in *NavMenu.razor.css* in de css-selector *.bi* de waarde voor *top* naar -0.75rem.

- Vervang in *NavMenu.razor* bij het Home-menu de css-class *bi-house-door-fill-nav-menu* dooor de class *bi-house-door-fill*.
- Voeg nu in de folder *Pages* een nieuwe component *ContactenLijst.razor* toe :

```
@page "/contactenlijst"

<h3>Contactenlijst</h3>

@if (Contacten != null) {

<table class="table">
    <thead>
        <tr>
            <th>Voornaam</th>
            <th>Naam</th>
            <th>Email</th>
        </tr>
    </thead>
    <tbody>
    @foreach (var c in Contacten) {
        <tr>
            <td>@c.Voornaam</td>
            <td>@c.Naam</td>
            <td>@c.Email</td>
        </tr>
    }
    </tbody>
</table>
}
else
{
    <p>Retrieving data...</p>
}

@code {

    [Inject]
    public ContactenService ContactenService { get; set; }

    public IEnumerable<Contact> Contacten { get; set; }

    protected override async Task OnInitializedAsync()
    {
        Contacten = (await ContactenService.GetContacten()).ToList();
    }
}
```

We bereiken deze component via de route */contactenlijst*. Op de page beelden we alle contacten af uit de property *Contacten.* Omdat het ophalen van deze contacten even op zich kan laten wachten zetten we de table binnen een if-block. Zolang de property *Contacten* null is mag er niks afgebeeld worden. Anders krijg je een runtimefout.

In de code-section voorzien we dus een Contacten-property. Deze vullen we op in de *OnInitialized-Async()*-method. We maken daarvoor gebruik van een contactenservice die we injecteren in de property *ContactenService*.

We missen wel nog enkele import-statements. Die kunnen we uiteraard toevoegen in de component maar het is een beter idee dit te doen in *_Imports.razor*.

- Voeg in *_Imports.razor* onderstaande using-statements toe :

```
@using Contacten.Web.Client.Services
@using Contacten.Models
```

Onze WebAPI zal aangesproken worden door de Blazor app. Die app wordt gehost op een andere locatie (URL) dan de WebAPI. Standaard wordt dit niet toegestaand. In de cursus WebAPI leerde je over CORS. Ook hier zullen we code moeten toevoegen in *Program.cs* om dit mogelijk te maken.

- Voeg in de WebAPI in *Program.cs* de CORS-service toe :

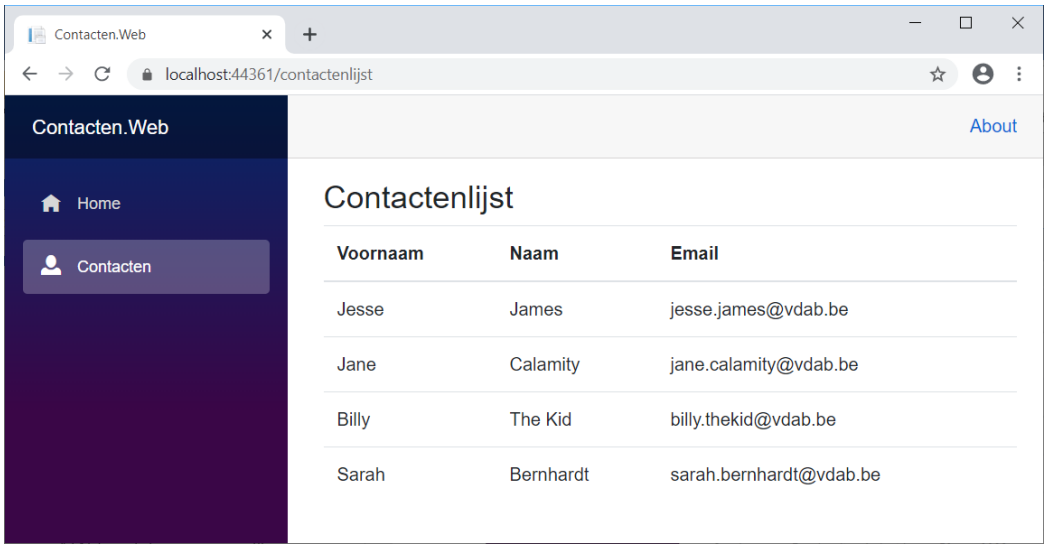
```
…
builder.Services.AddCors();
builder.Services.AddControllers();
…
app.UseCors(builder =>
    builder.AllowAnyOrigin().AllowAnyHeader().AllowAnyMethod());
app.UseAuthorization();
…
```

Vooraleer we dit uitproberen moeten we de projecten *Contacten.Data* én *Contacten.Web* als startupproject instellen.

- Klik in de Solution Explorer met de rechtermuistoets op de naam van de solution en kies *Configure Startup Projects…*
- Kies *Multiple startup projects* en selecteer de action *Start* bij *Contacten.Web* en *Contacten.Data.* Zorg ervoor dat *Contacten.Data* boven *Contacten.Web* staat.
- Klik op *OK*.

We kunnen de app nu uitproberen.

- Start de applicatie. De WebAPI wordt opgestart en na enkele seconden wordt de Swagger indexpage getoond. Ook de Blazor app wordt opgestart.
- Klik op het menuitem *Contacten.* Alle contacten worden in een tabel getoond :



14.4.2 Een detailpagina

We voegen een tweede component toe waarop alle details van één contact weergegeven worden.

In de serviceclass *ContactenService* voegen we eerst een extra method toe die op basis van de id van een contact een contact-object ophaalt uit de WebAPI en teruggeeft.

- Voeg in *ContactenService.cs* onderstaande method toe :

```
public async Task<Contact> GetContact(int id) {
    var contact = await httpClient.GetStringAsync($"api/contacten/{id}");
    var options = new JsonSerializerOptions
    {
        PropertyNameCaseInsensitive = true
    };
    return JsonSerializer.Deserialize<Contact>(contact, options);
}
```

We doen dus een API-call naar *api/contacten/* en vervolgens de id van het contact. Deze route is zo aangegeven in de API-controller.

- Voeg in de Blazor-app in het Client-project onderstaande extra component *ContactDetail.razor* toe :

```
@page "/contactdetail/{id:int}"

<div class="row justify-content-center">
    <div class="col-md-8 col-lg-6">
        <div class="card">
            <div class="card-header">
                <h1>@Contact.Voornaam @Contact.Naam</h1>
            </div>
            <div class="card-body">
                <h4><span class="oi oi-home" aria-hidden="true"
                    style="width:30px;"></span> @Contact.Adres</h4>
                <h4 style="margin-left:36px;">
                    @Contact.Postcode @Contact.Gemeente
                </h4>
                <h4><span class="oi oi-phone" aria-hidden="true"
                    style="width:30px;"></span> @Contact.Telefoon</h4>
                <h4><span class="oi oi-envelope-closed" aria-hidden="true"
                    style="width:30px;"></span> @Contact.Email</h4>
                <h4><span class="oi oi-calendar" aria-hidden="true"
                    style="width:30px;">
                    </span> @Contact.Geboortedatum.ToShortDateString()</h4>
            </div>
            <div class="card-footer text-center">
                <a href="/contactenlijst" class="btn btn-primary">Back</a>
                <a href="#" class="btn btn-primary">Edit</a>
                <a href="#" class="btn btn-danger">Delete</a>
            </div>
        </div>
    </div>
</div>

@code {
    [Parameter]
    public int Id { get; set; }

    public Contact Contact { get; set; } = new Contact();

    [Inject]
    public ContactenService ContactenService { get; set; }

    protected async override Task OnInitializedAsync()
    {
        Contact = await ContactenService.GetContact(Id);
    }

}
```

In de route naar deze page is er een parameter *id* die van het type int moet zijn. Vervolgens geven we de details van het contact weer in een Bootstrap-card. Voornaam en naam worden in de header van de card weergegeven, de overige gegevens in de body. In de footer plaatsen we een button *Back* die ons terug naar de lijst brengt. De buttons *Edit* en *Delete* implementeren we hierna.

In de code-section voorzien we een Parameter-property *Id*. Daarin zal de waarde van de id uit de URL in ingevuld worden. Aan de hand van deze id vragen we aan de geïnjecteerde contactenservice het contact op en wijzen deze toe aan de property *Contact.*

Tot slot voegen we in de contactenlijst aan de linkerkant een kolom toe met daarin een vergrootglas-icoon dat verwijst naar de detailspage.

- Voeg in *ContactenLijst.razor* in de lijst een extra kolom toe aan de table :

```
…
<table class="table">
    <thead>
        <tr>
            <th></th>
            <th>Voornaam</th>
            <th>Naam</th>
            <th>Email</th>
        </tr>
    </thead>
    <tbody>
    @foreach (var c in Contacten) {
        <tr>
            <td><a href="contactdetail/@c.ContactId"><span class=
                "bi bi-search" aria-hidden="true"></span></a></td>
            <td>@c.Voornaam</td>
            <td>@c.Naam</td>
            <td>@c.Email</td>
        </tr>
    }
    </tbody>
</table>
```

- Je kan opnieuw uitproberen : kies in het menu voor *Contacten.* Klik dan op een vergrootglas naast één van de contacten om de details ervan te zien.
- Klik op de Back-button om terug te keren naar de lijst.

14.4.3 Contactgegevens wijzigen

We implementeren nu de Edit-functie. Daarvoor voegen we in de ContactenService een method *UpdateContact()* toe.

- Voeg onderstaande method toe in *ContactenService.cs* :

```
public async Task<Contact> UpdateContact(Contact contact)
{
    HttpResponseMessage message = await httpClient.PutAsJsonAsync<Contact>(
        $"api/contacten/{contact.ContactId}", contact);
    if (message.IsSuccessStatusCode)
        return contact;
    else
        return null;
}
```

We gebruiken hier de HttpClient-method *PutAsJsonAsync<Contact>()* om de update-method uit de

WebAPI aan te roepen. Deze method heeft twee parameters : de URI van de updaterequest en het gewijzigde object. Om de method *PutAsJsonAsync* te kunnen gebruiken moet je wel de NuGet-package *Microsoft.AspNet.WebApi.Client* installeren.

- Installeer de package *Microsoft.AspNet.WebApi.Client* voor het project *Contacten.Web.Client*

We voegen nu een nieuwe component toe voor het updateformulier.

- Maak in de folder *Pages* een nieuwe component *EditContact.razor* en voeg onderstaande code toe.

```
@page "/editcontact/{id:int}"
@inject NavigationManager navmanager
<EditForm Model="@Contact" OnValidSubmit="UpdateContact">
    <DataAnnotationsValidator />
    <h3>Edit Contact</h3>
    <div class="row my-3">
        <label for="voornaam" class="col-sm-2 col-form-label">
            Voornaam
        </label>
        <div class="col-sm-6">
            <InputText id="voornaam" class="form-control"
                       @bind-Value="Contact.Voornaam" />
        </div>
    </div>
    <div class="row my-3">
        <label for="naam" class="col-sm-2 col-form-label">Naam</label>
        <div class="col-sm-6">
            <InputText id="naam" class="form-control"
                       @bind-Value="Contact.Naam" />
        </div>
    </div>
    <div class="row my-3">
        <label for="adres" class="col-sm-2 col-form-label">Adres</label>
        <div class="col-sm-6">
            <InputText id="adres" class="form-control"
                       @bind-Value="Contact.Adres" />
        </div>
    </div>
    <div class="row my-3">
        <label for="postcode" class="col-sm-2 col-form-label">Postcode</label>
        <div class="col-sm-2">
            <InputText id="postcode" class="form-control"
                       @bind-Value="Contact.Postcode" />
        </div>
    </div>
    <div class="row my-3">
        <label for="gemeente" class="col-sm-2 col-form-label">Gemeente</label>
        <div class="col-sm-6">
            <InputText id="gemeente" class="form-control"
                       @bind-Value="Contact.Gemeente" />
        </div>
    </div>
    <div class="row my-3">
        <label for="telefoon" class="col-sm-2 col-form-label">Telefoon</label>
        <div class="col-sm-2">
            <InputText id="telefoon" class="form-control"
                       @bind-Value="Contact.Telefoon" />
        </div>
    </div>
```

```
        <div class="row my-3">
            <label for="email" class="col-sm-2 col-form-label">Email</label>
            <div class="col-sm-6">
                <InputText id="email" class="form-control"
                           @bind-Value="Contact.Email" />
            </div>
        </div>
        <div class="row my-3">
            <label for="geboorte" class="col-sm-2 col-form-label">Geboortedatum
            </label>
            <div class="col-sm-2">
                <InputDate id="geboorte" class="form-control"
                           @bind-Value="Contact.Geboortedatum" />
            </div>
        </div>
        <ValidationSummary />
        <button type="submit" class="btn btn-primary">Save</button>
        <a href="/contactenlijst" class="btn btn-danger">Cancel</a>
    </EditForm>

    @code {
        public Contact Contact { get; set; } = new Contact();

        [Inject]
        public ContactenService ContactenService { get; set; }

        [Parameter]
        public int Id { get; set; }

        protected async override Task OnInitializedAsync()
        {
            Contact = await ContactenService.GetContact(Id);
        }

        private async void UpdateContact()
        {
            await ContactenService.UpdateContact(Contact);
            navmanager.NavigateTo($"/contactdetail/{Contact.ContactId}");
        }
    }
```

We overlopen even deze code. Bovenaan vinden we de route naar deze update-component. De id van het contact wordt als parameter meegegeven en komt zo in de property *Id* te staan.

Verderop in de code zullen we na een update terug naar de detailspage navigeren. Daarvoor hebben we een NavigationManager nodig. Deze wordt hier geïnjecteerd.

Dan volgt een EditForm. Het model van deze form is het te editeren contact uit de property *Contact.* Deze property wordt net als in de detailspage opgevuld in de method *OnInitializedAsync.* Daarvoor gebruiken we de geïnjecteerde ContactenService. De form wordt verwerkt door de method *UpdateContact()* uit de code-sectie.

We voegen een *DataAnnotationsValidator* toe zodat we validation kunnen toepassen. Eventuele foutboodschappen worden onderaan de form weergegeven in de *ValidationSummary*-component.

Vervolgens gebruiken we een aantal *InputText*-componenten en één *InputDate*-component om de te editeren velden weer te geven. De id van het contact hoort daar niet bij en geven we ook niet weer. Deze input-componenten worden telkens aan één veld gebonden via het @bind-Value attribuut.

Onderaan de form vinden we twee buttons : een submit-button die de method *UpdateContact()* activeert en een Cancel-button die eigenlijk een link is naar de detailspage. In de method *UpdateContact()* roepen we de updatemethod uit de service op en keren we terug naar de detailspage. Je zou eventueel een foutboodschap kunnen laten zien wanneer de updatemethod uit de serviceclass null oplevert.

Rest ons nog de link naar de editpage toe te voegen in de detailspage.

- Wijzig in *ContactDetail.razor* de code in de cardfooter als volgt :

```
<div class="card-footer text-center">
    <a href="/contactenlijst" class="btn btn-primary">Back</a>
    <a href="/editcontact/@Contact.ContactId" class="btn btn-primary">Edit</a>
    <a href="#" class="btn btn-danger">Delete</a>
</div>
```

- Probeer uit.

14.4.4 Een contact verwijderen

Na het wijzigen van een contact implementeren we nu de delete-feature.

- We voegen een method *DeleteContact()* toe in *ContactenService.cs* :

```
public async Task DeleteContact(int id)
{
    await httpClient.DeleteAsync($"api/contacten/{id}");
}
```

Wat de UI betreft zijn er nu verschillende opties. In een desktopapplicatie zou een klik op de Delete-knop wellicht een popupvenster naar voor brengen waarin je de delete-actie kan bevestigen. Een ASP.NET webapplicatie zou een nieuwe page tonen met ook daarop de mogelijkheid om te bevestigen.

In een SPA moeten we de gebruiker het gevoel geven dat hij/zij slechts over één pagina beschikt. Daarom gaan we de buttons *Back, Edit* en *Delete* na een klik op de *Delete*-button verbergen en vervangen door een boodschap en de knoppen *Yes* en *No.*

De page heeft dus twee statussen : ofwel worden er 3 buttons getoond (normale situatie) ofwel een bevestigingsboodschap en *Yes* en *No* buttons (delete-bevestiging). De status van de page bewaren we in een boolean.

- Voeg in de component *ContactDetail* een boolean-member toe :

```
@code {

    private bool deletePressed = false;
    …
}
```

In de layout tonen we nu ofwel de 3 buttons, ofwel de boodschap en de knoppen *Yes* en *No*.

- Wijzig de layout voor de card-footer als volgt :

```
<div class="card-footer text-center">
    @if (!deletePressed)
    {
        <div>
```

```
                <a href="/contactenlijst" class="btn btn-primary">Back</a>
                <a href="/editcontact/@Contact.ContactId"
                   class="btn btn-primary">Edit</a>
                <button class="btn btn-danger">
                    Delete
                </button>
            </div>
        } else {
            <div>
                <p>Delete @Contact.Voornaam @Contact.Naam?</p>
                <button class="btn btn-danger">Yes</button>
                <button class="btn btn-primary">No</button>
            </div>
        }
    </div>
```

De defaultwaarde van de boolean *deletePressed* zetten we op false. Op die manier worden standaard de 3 buttons getoond.

Bij het klikken op de Delete-button moet deze boolean omgekeerd worden. We maken daarvoor een method die bij een onclick-event wordt geactiveerd.

- Voeg in de code-section onderstaande method toe :

```
private void ToggleDelete()
{
    deletePressed = !deletePressed;
}
```

- Wijzig de code voor de Delete button :

```
<button class="btn btn-danger" @onclick="ToggleDelete">Delete</button>
```

We voegen nu ook de nodige functionaliteit toe aan de *Yes* en *No* buttons. De *No* button is het eenvoudigst. Deze moet er enkel voor zorgen dat de 3 buttons terug zichtbaar worden en daarvoor hebben we reeds een method.

- Wijzig de code voor de *No*-button :

```
<button class="btn btn-primary" @onclick="ToggleDelete">No</button>
```

Een klik op de *Yes*-button moet het contact verwijderen en de volledige contactenlijst laten zien. We voegen daarvoor de nodige code toe in een method *DeleteContact().*

- Voeg in de code-section onderstaande method toe:

```
private async void DeleteContact()
{
    await ContactenService.DeleteContact(Id);
    navmanager.NavigateTo("/contactenlijst");
}
```

In deze method roepen we de service-method *DeleteContact()* op om het contact definitief te verwijderen. Daarna gaan we terug naar de contactenlijst. Om dat te kunnen doen moeten we een navigationmanager injecteren.

- Voeg bovenaan de component onderstaande code toe :

```
@inject NavigationManager navmanager
```

We kunnen de DeleteContact-method nu koppelen aan het click-event van de *Yes*-button.

- Wijzig de code voor de *Yes*-button als volgt :

```
<button class="btn btn-primary" @onclick="DeleteContact">Yes</button>
```

- Je kan het verwijderen van een contact nu uitproberen.

14.4.5 Een contact toevoegen

Als laatste CRUD-operatie is het toevoegen aan de beurt. Op de contactenlijst-page voegen we onderaan een *Create*-button toe. Deze brengt ons op een page waarin we de nodige velden invullen.

Eigenlijk doen we hier min of meer hetzelfde als bij het wijzigen van een contact met dit verschil dat we hier starten vanaf een blanco record. We kunnen dus een groot deel van de *EditContact*-component hergebruiken.

Laten we beginnen met onder de contactenlijst een *Create*-knop toe te voegen.

- Voeg in *ContactenLijst.razor* onderstaande code toe net onder de table-endtag :

```
<a href="/editcontact" class="btn btn-primary">Create</a>
```

De link verwijst naar de *EditContact*-component maar we kunnen geen id meegeven. Opdat de opgegeven component toch zou weergegeven worden moeten we in *EditContact.razor* een extra route toevoegen, maar dan zonder parameter.

- Voeg in *EditContact.razor* bovenaan de page onderstaande extra route toe :

```
@page "/editcontact"
```

De EditContact-component wordt nu voor twee verschillende scenario's gebruikt : voor het wijzigen van een bestaand contact en voor het invullen van de gegevens van een nieuw contact. In de method *OnInitializedAsync()* wordt nu het contact opgezocht dat hoort bij de parameter *Id*. Bij het toevoegen van een nieuw contact hebben we geen dergelijke parameter en hoeft dit niet te gebeuren. De property *Contact* bevat een leeg contact-object en dat is prima als start.

- Wijzig de code in *OnInitializedAsync()* als volgt :

```
protected async override Task OnInitializedAsync()
{
    if (Id != 0) Contact = await ContactenService.GetContact(Id);
}
```

Na het invullen of updaten van de contactgegevens kan de gebruiker op *Save* of *Cancel* drukken. De code voor het bewaren van een nieuw contact is anders dan bij het updaten van een bestaand contact. We passen de code in de method *UpdateContact()* aan.

- Wijzig de code in *UpdateContact()* als volgt :

```
private async void UpdateContact()
{
    if (Id != 0) {
        await ContactenService.UpdateContact(Contact);
        navmanager.NavigateTo($"/contactdetail/{Contact.ContactId}");
    }
    else {
        await ContactenService.AddContact(Contact);
        navmanager.NavigateTo($"/contactenlijst");
    }
}
```

Ook hier bepalen we op basis van de waarde van Id of we een toevoegactie of een updateactie moeten ondernemen. Bij de toevoeging van een nieuw contact gebruiken we een method *AddContact()* uit de contactenservice om het contact op te slaan.

We voegen deze method nu toe in de contactenservice.

- Voeg in *ContactenService.cs* onderstaande method toe :

```
public async Task<Contact> AddContact(Contact contact)
{
    HttpResponseMessage message = await httpClient.PostAsJsonAsync<Contact>(
        $"api/contacten", contact);
    if (message.IsSuccessStatusCode)
        return contact;
    else
        return null;
}
```

Om een contact toe te voegen gebruiken we de HttpClient-method *PostAsJsonAsync<Contact>().* Parameters zijn de URI en het nieuwe Contact-object.

Ook de *Cancel*-button moet anders reageren bij een update dan bij een toevoeging. Na een gecancelde update willen we gewoon op de detailpage blijven van het contact. Na het cancelen van een toevoeging willen we terug naar de contactenlijst.

- Pas in de layout van de EditContact-component de code voor de *Cancel*-button als volgt aan :

```
<a href=@(Id == 0 ? "/contactenlijst" : $"/contactdetail/{Contact.ContactId}")
   class="btn btn-danger">Cancel</a>
```

Om de link naar de juiste route te laten wijzen gebruiken we de conditional operator ?.

Ook de header in EditContact.razor kunnen we op een gelijkaardige manier aanpassen. Bij het editeren van een contact moet de header *Edit Contact* zijn, bij het toevoegen *Create Contact.*

- Wijzig in EditContact.razor de <h3>-tag als volgt :

```
<h3>@(Id == 0 ? "Create Contact" : "Edit Contact")</h3>
```

- Je kan alle CRUD-operaties nu uittesten.

14.5 Virtualization en lazy loading

De contactenlijst-applicatie werkt nu prima maar heeft nog één groot nadeel. Al onze contacten worden in één keer opgeladen en allemaal in het geheugen van de client bewaard. Vooral bij smartphones is dit niet zo interessant. Die hebben soms een beperkter geheugen. Er wordt wellicht ook veel te veel data ingeladen wat dan weer niet interessant is voor jouw 4G/5G-abonnement. Het opladen van alle contacten kan trouwens meerdere seconden duren wat niet toelaatbaar is qua responstijd.

14.5.1 Extra contacten

Om te bewijzen dat onze huidige oplossing voor verbetering vatbaar is voegen we een aantal extra contacten toe.

- Start SSMS.
- Open het script *extraContacten.sql* en voer het uit.

Je beschikt nu over iets meer dan 1000 contacten.

- Start de applicatie en kies in het menu het menuitem *Contacten.*

Je krijgt nu normaal pas na meerdere seconden de lijst te zien. Op mijn toestel is dit ongeveer 3 seconden. Qua responstijd is dit niet aanvaardbaar.

- Stop de applicatie nog niet maar laat ze nog even open staan.

14.5.2 Virtualization

We lossen het performantieprobleem op in twee stappen. In een eerste stap verbeteren we de rendering van de page. Op dit moment wordt er een page samengesteld met daarin een table met meer dan 1000 rijen. Hiervan zijn er slechts een paar zichtbaar voor de gebruiker.

- Druk op F12 om de Developer Tools te openen.
- Ga naar het tabblad *Elements*.
- Klap de tags <body>, , <main>, <article>, <table> en <tbody> open :

```
Elements   Console   Sources   Network   >>
<!DOCTYPE html>
<html lang="en" style="--blazor-load-percentage: 100%; --blazo
-text: "100%";">
▶ <head> ··· </head>
▼ <body>
    <!--!-->
    <!--!-->
    <!--!-->
    <!--!-->
    <!--!-->
  ▼ <div class="page" b-0ttgda90q4> flex
    ▶ <div class="sidebar" b-0ttgda90q4> ··· </div>
      <!--!-->
    ▼ <main b-0ttgda90q4>
        <!--!-->
      ▶ <div class="top-row px-4" b-0ttgda90q4> ··· </div> flex
      ▼ <article class="content px-4" b-0ttgda90q4>
          <!--!-->
          <!--!-->
          <h3>Contactenlijst</h3>
        ▼ <table class="table">
            <!--!-->
          ▶ <thead> ··· </thead>
··· ▼ <tbody> == $0
            ▶ <tr> ··· </tr>
```

De table bevat zó veel <tr>-tags dat ze zelfs niet allemaal worden getoond in de Developer Tools.

```
▶ <tr>…</tr>
▶ <tr>…</tr>
  Show All Nodes (504 More)
</tbody>
</table>
<!--!-->
```

Blazor beschikt over een component Virtualize die ervoor zorgt dat er slechts een beperkter aantal elementen gerenderd worden. Welke elementen dit zijn hangt o.a. af van de vertikale scrollpositie in de lijst en de hoogte van één element.

- Wijzig de code in ContactenLijst.razor als volgt :

```
…
<tbody>
    <Virtualize Items="Contacten" TItem="Contact" Context="c">
        <tr>
            <td>
                <a href="contactdetail/@c.ContactId">
                    <span class="bi bi-search" aria-hidden="true"></span>
                </a>
            </td>
            <td>@c.Voornaam</td>
            <td>@c.Naam</td>
            <td>@c.Email</td>
        </tr>
    </Virtualize>
</tbody>
…
@code {
    …
    public ICollection<Contact> Contacten { get; set; }
    …
}
```

De component Virtualize heeft 3 parameters. Via Items geef je de lijst met contacten door. TItem geeft het type aan van een contact. Om binnen de Virtualize-tag te verwijzen naar het weer te geven contact gebruik je de contextvariabele *c*. Deze naam is uiteraard vrij te kiezen.

De structuur van de Virtualize component is dus gelijkaardig aan de TableTemplate op p. 35.

In de code-section wijzigen we het type van Contacten naar een ICollection<Contact>. Dit moet omdat de parameter *Items* van de component *Virtualize* dit type verwacht.

Om onze applicatie nog wat meer op de proef te stellen en om te zien waar we ons ongeveer bevinden in de lijst voegen we nog de contactid en de geboortedatum toe in de tabel.

- Wijzig de layoutcode in *ContactenLijst.razor* zodat de contactid en de geboortedatum ook zichtbaar zijn in de tabel.

	Id	Voornaam	Naam	Email	Geboortedatum
	1	Jesse	James	jesse.james@vdab.be	01-01-1966

- Start de applicatie en kies het menuitem *Contacten.*

De lijst laat opnieuw enkele seconden op zich wachten. Wanneer je echter via de Developer Tools naar de gerenderde pagina kijkt merk je dat de tabel deze keer slechts een 30-tal rijen bevat.

- Druk opnieuw op F12 voor de Developer Tools en ga naar het tabblad *Elements*.
- Klap de html-structuur open totdat je de inhoud van de <tbody>-tag ziet.
- Klap één <tr>-tag open. Bekijk de inhoud.
- Scroll doorheen de page. De inhoud van de <tr>-tag verandert. Het aantal rijen is nog steeds dezelfde.
- Verklein jouw browservenster zodat het nog slechts de helft van jouw scherm inneemt.
- Refresh de pagina. Deze keer worden er nog hoogstens een 20-tal rijen gerenderd.

De Virtualize-component rendert dus slechts een beperkt aantal rijen. Het bepaalt het aantal rijen aan de hand van de beschikbare ruimte en de hoogte van één rij. We kunnen deze berekening op twee manieren beïnvloeden. Een Virtualize-component beschikt namelijk over twee extra attributen: **ItemSize** en **OverscanCount**.

ItemSize is de hoogte van één rij en heeft een standaardwaarde van 50 pixels. Met het attribuut OverscanCount bepaal je hoeveel extra elementen er moeten gerenderd worden voor en na de zichtbare elementen. Dit zijn er standaard 3.

Voorbeeld :

```
<Virtualize Items="Contacten" TItem="Contact" Context="c"
            ItemSize="75" OverscanCount="1">
```

14.5.3 Lazy loading

Ons grootste probleem, het trage laden van de contacten, is echter nog steeds niet opgelost. Daarvoor gebruiken we de lazy loading techniek. In plaats van alle contacten in één keer te laden zullen we enkel de zichtbare contacten ophalen uit de database.

Vooraleer we dit oplossen tonen we nog even aan hoeveel data er wordt ingeladen, telkens we de Contactenpage tonen in de browser.

- Herstart de applicatie.
- Druk op F12 voor de Developer Tools.
- Activeer de tab *Network.*
- Klik op het menuitem *Contacten* om de contactenlijst te zien. Er wordt meer dan 200kB aan gegevens binnengeladen.

contacten	200	fetch	polyfills.ts:120	222 kB

- Kies in het menu voor Home en daarna opnieuw Contacten. Er wordt opnieuw meer dan 200kB ingeladen.

Met lazy loading zullen we telkens slechts enkele kB binnenladen. Een fractie dus van wat we nu ophalen.

In de Virtualize component vervangen we het attribuut *Items* door een attribuut *ItemsProvider*. Met dit attribuut geven we de method aan die enkel de nodige records zal ophalen (en niet alle records). Dit zijn dus de zichtbare records plus eventueel enkele extra records naar gelang de waarde van het attribuut OverscanCount.

Een tweede aanpassing in de component ContactenLijst.razor is de membervariabele *Contacten.* Die gaan we niet langer gebruiken. We voegen wel een membervariabele *totaalContacten* toe. We initialiseren deze variabele in de method OnInitializedAsync en gebruiken ze vervolgens in de ItemsProvider. Die moet immers weten hoeveel records er in totaal zijn. We zullen *totaalContacten* trouwens ook gebruiken om te bepalen of de wachttekst *Retrieving data…* al dan niet moet worden afgebeeld.

- Wijzig de code in ContactenLijst.razor als volgt :

```
@page "/contactenlijst"
<h3>Contactenlijst</h3>
```

```
@if (totaalContacten != 0)
{
    <table class="table">
        <thead>
            <tr>
                <th></th>
                <th>Id</th>
                <th>Voornaam</th>
                <th>Naam</th>
                <th>Email</th>
                <th>Geboortedatum</th>
            </tr>
        </thead>
        <tbody>
            <Virtualize ItemsProvider="@LoadContacten"
                        TItem="Contact" Context="c">
                <tr>
                    <td>
                        <a href="contactdetail/@c.ContactId">
                            <span class="bi bi-search" aria-hidden="true"></span>
                        </a>
                    </td>
                    <td>@c.ContactId</td>
                    <td>@c.Voornaam</td>
                    <td>@c.Naam</td>
                    <td>@c.Email</td>
                    <td>@c.Geboortedatum.ToShortDateString()</td>
                </tr>
            </Virtualize>
        </tbody>
    </table>
    <a href="/editcontact" class="btn btn-primary">Create</a>
}
else
{
    <p>Retrieving data...</p>
}

@code {

    [Inject]
    public ContactenService ContactenService { get; set; }

    private int totaalContacten;

    private async ValueTask<ItemsProviderResult<Contact>> LoadContacten(
        ItemsProviderRequest request)
    {
        var aantalContacten =
            Math.Min(request.Count, totaalContacten - request.StartIndex);
        var contacten = await ContactenService.GetContactenSubset(request.StartIndex,
            aantalContacten);
        return new ItemsProviderResult<Contact>(contacten, totaalContacten);
    }

    protected override async Task OnInitializedAsync()
    {
        totaalContacten = (await ContactenService.GetAantalContacten());
    }
}
```

In de layout zien we dus een wijziging in het if-statement bovenaan. Daar wordt deze keer de variabele *totaalContacten* gebruikt. Pas als deze een waarde gekregen heeft laten we de table zien. Verder is in de Virtualize component het attribuut *Items* vervangen door *ItemsProvider.*

In de code-section is de membervariabele *Contacten* vervangen door een int *totaalContacten.* We geven deze variabele in *OnInitializedAsync()* een waarde door de servicemethod *GetAantalContacten* op te roepen. Deze method voegen we zodadelijk toe.

Het meeste werk gebeurt echter in de method *LoadContacten.* Deze method zal uitgevoerd worden telkens wanneer er records moeten afgebeeld worden die nog niet gerenderd zijn. In deze method wordt eerst bepaald hoeveel records er moeten opgehaald worden uit de database. De variabele *request.Count* is het berekend aantal records volgens de waarde van *ItemSize* en van *OverscanCount*. In sommige gevallen is dit te veel. Bijvoorbeeld als we net niet aan het einde van de lijst zitten en er nog slechts een paar records extra moeten opgehaald worden. Of wanneer er minder records in de database zitten dan het aantal dat theoretisch kan afgebeeld worden.

Vervolgens halen we een subset van contactenrecords op. Dit is een aantal records vanaf een bepaalde beginindex. We gebruiken hiervoor een servicemethod *GetContactenSubset*. Deze method moeten we nog schrijven.

Het resultaat van de ItemsProvider is een *ItemsProviderResult<Contact>*. Deze krijgt de opgehaalde subset contactenrecords en het totaal aantal contacten mee.

We schrijven nu de twee extra servicemethods *GetAantalContacten* en *GetContactenSubset.*

- Voeg in de ContactenService onderstaande methods toe :

```
public async Task<int> GetAantalContacten()
{
    var aantalContactenString = await
        httpClient.GetStringAsync("api/aantalcontacten");
    return JsonSerializer.Deserialize<int>(aantalContactenString)!;
}

public async Task<IEnumerable<Contact>> GetContactenSubset(int from,
    int number)
{
    var contactenString = await
        httpClient.GetStringAsync($"api/contacten/{from}/{number}");
    var options = new JsonSerializerOptions
    {
        PropertyNameCaseInsensitive = true
    };
    return JsonSerializer.Deserialize<Contact[]>(contactenString, options)!;
}
```

Om het totale aantal contacten te bepalen zullen we een nieuwe API-method schrijven met als route *api/aantalcontacten*. Deze method levert een int op.

We zullen ook een tweede API-method toevoegen die lijkt op de method die alle contacten ophaalt. Het verschil is dat deze extra method twee parameters heeft : het beginpunt en het aantal op te halen contacten.

We voegen deze API-methods nu toe in de ContactenController.

- Voeg onderstaande methods toe in ContactenController.cs :

```
[Route("api/[controller]")]
[ApiController]
public class ContactenController : ControllerBase
{
    …
    [HttpGet]
    [Route("/api/aantalcontacten")]
    public async Task<ActionResult> GetAantalContacten()
    {
        try
        {
            return Ok(await contactenRepository.GetAantalContacten());
        }
        catch (Exception)
        {
            return StatusCode(StatusCodes.Status500InternalServerError,
                "Database error");
        }
    }

    [HttpGet("{from:int}/{number:int}")]
    public async Task<ActionResult> GetContactenSubset(int from, int number)
    {
        try
        {
            return Ok(await contactenRepository.GetContactenSubset(from, number));
        }
        catch (Exception)
        {
            return StatusCode(StatusCodes.Status500InternalServerError,
                "Database error");
        }
    }
}
```

De action method *GetAantalContacten* heeft een wat afwijkende route. Vergeet hier de begin-slash niet ! Deze overschrijft de (deel-)route die bij de controller is aangegeven. De method zelf roept een repo-method *GetAantalContacten* op.

De action method *GetContactenSubset* gebruikt een gelijknamige method uit de contactenrepository.

We voegen de twee methods toe in de interface *IContactenRepository* en in de implementatie *ContactenRepository.*

- Wijzig de interface *IContactenRepository* als volgt :

```
public interface IContactenRepository
{
    …
    Task<int> GetAantalContacten();
    Task<IEnumerable<Contact>> GetContactenSubset(int from, int number);
}
```

- Implementeer deze extra methods in ContactenRepository.cs :

```
public async Task<int> GetAantalContacten()
{
    return await context.Contacten.CountAsync();
}
```

```
public async Task<IEnumerable<Contact>> GetContactenSubset(int from,
    int number)
{
    return await context.Contacten.Skip(from).Take(number).ToListAsync();
}
```

Je kan dit nu uitproberen.

- Start de app en kies het menuitem *Contacten.* Na enkele tellen zie je de contactenlijst.
- Druk op F12 en controleer op het tabblad *Elements* uit hoeveel rijen de table bestaat.
- Activeer het tabblad *Network* en scroll even door de lijst. Er wordt slechts een paar kB aan gegevens opgehaald.

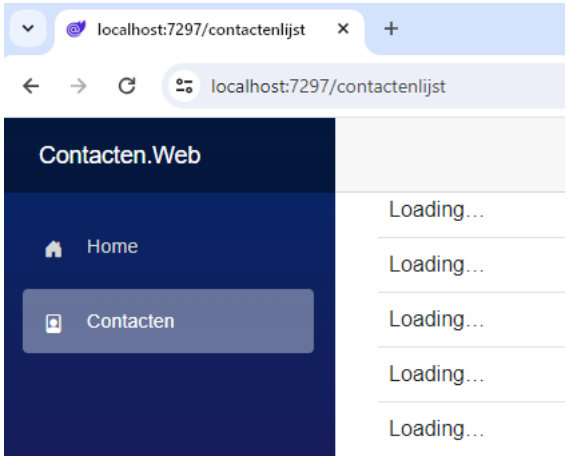
We brengen nog een allerlaatste esthetische verbetering aan in de contactenlijst. Wanneer je een eind vooruit of achteruit bladert in de lijst is een stuk van het scherm soms wit. In afwachting dat de weer te geven contacten gerenderd zijn kunnen we een wachttekst zoals *Loading…* weergeven. We gebruiken hiervoor een <Placeholder>-tag. De contactinformatie stop je in een <ItemContent>-tag.

- Voeg aan de <Virtualize>-component een childelement <Placeholder> toe en stop de tablerow met de contactgegevens in een <ItemContent>-element :

```
<Virtualize ItemsProvider="@LoadContacten" TItem="Contact" Context="c">
   <ItemContent>
       <tr>
           …
       </tr>
   </ItemContent>
   <Placeholder>
       <tr>
           <td colspan="6">Loading&hellip;</td>
       </tr>
   </Placeholder>
</Virtualize>
```

Let er wel op dat de code binnen de <Placeholder>-tag dezelfde structuur heeft als deze binnen de <ItemContent>-tags. In ons geval moet het dus ook een <tr>-tag zijn met 6 <td> subtags of een equivalent ervan door columns te mergen via het attribuut colspan. Doe je dit niet dan wordt er misschien ongeldige html gerenderd.

- Probeer opnieuw uit. Scroll snel doorheen de lijst. Je ziet onderstaand effect.



15 Voorbeeldoplossingen oefeningen

15.1 Dobbelen

```
@page "/dobbelen"

<h1>Dobbelen</h1>
<button class="btn btn-primary m-3" @onclick="GooiStenen">Gooi de stenen</button>
<div class="mb-3">
    @if (steen1 != 0 && steen2 != 0) {
        <img src=@($"/images/{steen1}.png") width="48" />
        <img src=@($"/images/{steen2}.png") width="48" />
    }
</div>
<p class="alert alert-info">Jouw score : @currentScore</p>

@code {
    private int currentScore = 0;
    private int steen1;
    private int steen2;
    private void GooiStenen() {
        Random random = new Random();
        steen1 = random.Next(1, 7);
        steen2 = random.Next(1, 7);
        currentScore = steen1 + steen2;
    }
}
```

15.2 Dobbelen versie 2

```
@page "/dobbelen"
@page "/dobbelen/{aantal:int}"
<h1>Dobbelen</h1>
<h3>We spelen met @Aantal stenen</h3>
<button class="btn btn-primary m-3" @onclick="GooiStenen">Gooi de stenen</button>
<div class="mb-3">
    @if (currentScore != 0) {
        foreach (int score in stenen) {
            <img src=@($"/images/{score}.png") width="48" />
        }
    }
</div>
<p class="alert alert-info">Jouw score : @currentScore</p>

@code {

    private int currentScore = 0;
    private List<int> stenen = new List<int>();

    [Parameter]
    public int Aantal { get; set; }

    private void GooiStenen()
    {
        currentScore = 0;
        stenen.Clear();
        Random random = new Random();
        for (var index = 0; index < Aantal; index++) {
            var ogen = random.Next(1, 7);
            stenen.Add(ogen);
            currentScore += ogen;
        }
    }
```

```
    protected override void OnInitialized()
    {
        base.OnInitialized();
        if (Aantal == 0) Aantal = 2;
    }
}
```

15.3 Boekenlijst

Boek.cs

```
public class Boek
{
    public string Auteur { get; set; }
    public string Titel { get; set; }
    public string ISBN { get; set; }
    public string Taal { get; set; }
    public string Afwerking { get; set; }
    public string UitgifteDatum { get; set; }
    public string Afbeelding { get; set; }
    public int Paginas { get; set; }
    public string Beschrijving { get; set; }
}
```

Tables.razor

```
@page "/tables"
<TableTemplate Items="boeken" TItem="Boek" Context="Boek">
    <TableHeader>
        <h1>Boekenlijst</h1>
    </TableHeader>
    <RowTemplate>
        <div class="links">
            <img src="/images/@Boek.Afbeelding" height="200" width="140"/>
        </div>
        <div class="rechts">
            <h3>@Boek.Auteur</h3>
            <h2>@Boek.Titel</h2>
            <p>@Boek.Taal | @Boek.Afwerking | @Boek.ISBN | @Boek.UitgifteDatum |
@Boek.Paginas pagina's</p>
            <h6>@Boek.Beschrijving</h6>
        </div>
    </RowTemplate>
</TableTemplate>

@code {
    public IEnumerable<Boek> boeken { get; set; } = new List<Boek>() {
        new Boek {
            Auteur = "J.K. Rowling", Titel = "Harry Potter en de steen der wijzen",
            Afwerking = "Paperback", Paginas = 301,
            ISBN = "9789061699767", Taal = "Nederlands",
            UitgifteDatum = "januari 2022", Afbeelding = "SteenDerWijzen.jpg",
            Beschrijving = "Harry Potter is een doodgewone, maar ongelukkige jongen
die sinds de dood van zijn ouders bij zijn saaie en hardvochtige oom en tante woont,
in de bezemkast onder de trap." },
        new Boek {
            Auteur = "Fjodor Dostojevski", Titel = "Schuld en boete",
            Afwerking = "Paperback", Paginas = 286,
            ISBN = "9789031505128", Taal = "Nederlands",
            UitgifteDatum = "september 2008", Afbeelding = "SchuldEnBoete.jpg",
            Beschrijving = "Schuld en boete is de geschiedenis van de geestelijk en
sociaal onderkomen student Raskoljnikow, die door een 'Napoleontische' daad zijn
```

```
gevoel van eigenwaarde en onafhankelijkheid tracht te handhaven en een moordenaar
wordt. Later komt hij door zijn liefde voor het gevallen meisje Sonja, die verheffend
op hem werkt, tot vrijwillige schuldbetekenis." },
        new Boek {
            Auteur = "Hugo Claus", Titel = "Het verdriet van België",
            Afwerking = "Paperback", Paginas = 720,
            ISBN = "9789403108100", Taal = "Nederlands",
            UitgifteDatum = "februari 2018", Afbeelding = "VerdrietVanBelgie.jpg",
            Beschrijving = "Het verdriet van België omspant de periode 1939-1947:
vrede, oorlog en vrede. Louis Seynaeve is elf jaar en leerling op een nonneninternaat.
Verwarring, hunkering en bedrog vormen zijn jongensjaren in de schoot van een
kleurrijke en verbazingwekkende familie." }
    };
}
```

app.css

```
.links {
    float:left;
    margin-right:20px;
}
.rechts {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
}
```

15.4 Dobbelen versie 3

15.4.1 Met een partial class

Dobbelen.razor

```
@page "/dobbelen"
@page "/dobbelen/{aantal:int}"
<h1>Dobbelen</h1>
<h3>We spelen met @Aantal stenen</h3>
<button class="btn btn-primary m-3" @onclick="GooiStenen">Gooi de stenen</button>
<div class="mb-3">
    @if (currentScore != 0) {
        foreach (int score in stenen) {
            <img src=@($"/images/{score}.png") width="48" />
        }
    }
</div>
<p class="alert alert-info">Jouw score : @currentScore</p>
```

Dobbelen.razor.cs

```
using Microsoft.AspNetCore.Components;

namespace BlazorApp.Pages
{
    public partial class Dobbelen
    {
        …
    }
}
```

15.4.2 Met een baseclass

Dobbelen.razor

```
@page "/dobbelen"
@page "/dobbelen/{aantal:int}"
@inherits DobbelenBase
<h1>Dobbelen</h1>
…
```

DobbelenBase.cs

```
using Microsoft.AspNetCore.Components;

namespace BlazorApp.Pages
{
    public class DobbelenBase : ComponentBase
    {
        protected int currentScore = 0;
        protected List<int> stenen;

        [Parameter]
        public int Aantal { get; set; }

        protected void GooiStenen()
        {
            …
        }

        protected override void OnInitialized()
        {
            …
        }
    }
}
```

15.5 Cascading value

Cascade.razor

```
…
<CascadingValue Name="DePersoon" Value="@Depersoon">
    <CascadingValue Name="HetGetal" Value="@Getal">
        <CascadingValue Value="@Woord">
            <SubComponent />
        </CascadingValue>
    </CascadingValue>
</CascadingValue>

@code {
    private Persoon Depersoon =
        new Persoon { Id = 1, Voornaam = "Steven", Familienaam = "Lucas" };
    …
}
```

SubSubComponent.razor

```
…
<p>@Persoon?.Voornaam @Persoon?.Familienaam</p>
…
```

```
@code {
    …
    [CascadingParameter(Name = "DePersoon")]
    public Persoon? Persoon { get; set; }
}
```

Opmerking : het *Name*-attribuut kan in beide componenten in principe weggelaten worden.

15.6 Data binding

```
@page "/binding"
<h3>Binding</h3>
<textarea rows="3" cols="20" @bind="ingetikteTekst" @bind:event="oninput"></textarea>
<p>Lengte van de tekst: @lengte()</p>
@code {
    private string ingetikteTekst = "voorbeeld";
    private int lengte()
    {
        return ingetikteTekst.Length;
    }
}
```

15.7 Protected Browser Storage

```
@page "/counter"
@using Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage
@inject ProtectedLocalStorage localStore
…
@code {
    private int currentCount = 0;
    private async Task IncrementCountAsync()
    {
        currentCount++;
        await localStore.SetAsync("counter", currentCount);
    }
    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            try {
                var result = await localStore.GetAsync<int>("counter");
                if (result.Success) {
                    currentCount = result.Value;
                    StateHasChanged();
                }
            }
            catch (Exception ex) {
                currentCount = 0;
            }
        }
    }
}
```

15.8 EditForm

Sollicitant.cs

```
public class Sollicitant
{
    [Required(ErrorMessage = "{0} is een verplicht veld")]
    [StringLength(30, ErrorMessage =
        "Gebruik maximum {1} tekens voor het veld {0}")]
    public string Naam { get; set; }
    [Required(ErrorMessage = "{0} is een verplicht veld")]
    [StringLength(30, ErrorMessage =
        "Gebruik maximum {1} tekens voor het veld {0}")]
    public string Voornaam { get; set; }
    [GeboortedatumValidator(ErrorMessage =
        "Geboortedatum moet in het verleden liggen")]
    public DateTime GeboorteDatum { get; set; }
    public bool Rijbewijs { get; set; }
    [Required(ErrorMessage = "{0} is een verplicht veld")]
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
    public Diploma Diploma { get; set; }
    [Required(ErrorMessage = "{0} is een verplicht veld")]
    public string Functie { get; set; }
}
```

Diploma.cs

```
public enum Diploma
{
    geen,
    middelbaaronderwijs,
    hogeronderwijsbuitenuniv,
    hogeronderwijsuniv
}
```

Sollicitatieformulier.razor

```
@page "/solliciteren"
<h1>Sollicitatieformulier</h1>
<EditForm Model="@sollicitant" OnValidSubmit="HandleValidSubmit">
    <DataAnnotationsValidator />
    <div class="row my-3">
        <label for="naam" class="col-sm-2 col-form-label">Naam</label>
        <div class="col-sm-4">
            <InputText id="naam" @bind-Value="sollicitant.Naam" class="form-control" />
        </div>
    </div>
    <div class="row my-3">
        <label for="voornaam" class="col-sm-2 col-form-label">Voornaam</label>
        <div class="col-sm-4">
            <InputText id="voornaam" @bind-Value="sollicitant.Voornaam"
                       class="form-control" />
        </div>
    </div>
    <div class="row my-3">
        <label for="geboortedatum" class="col-sm-2 col-form-label">Geboortedatum</label>
        <div class="col-sm-2">
            <InputDate id="geboortedatum" @bind-Value="sollicitant.GeboorteDatum"
                       class="form-control" />
        </div>
    </div>
```

```
    <div class="row my-3">
        <label for="email" class="col-sm-2 col-form-label">Email</label>
        <div class="col-sm-4">
            <InputText id="email" @bind-Value="sollicitant.Email"
                class="form-control" />
        </div>
    </div>
    <div class="row my-3">
        <label for="rijbewijs" class="col-sm-2 col-form-label">Rijbewijs</label>
        <div class="col-sm-1 mt-2">
            <InputCheckbox id="rijbewijs" @bind-Value="sollicitant.Rijbewijs" />
        </div>
    </div>
    <div class="row my-3">
        <div class="col-sm-2">Diploma</div>
        <div class="col-sm-4 form-check ml-3">
            <InputRadioGroup @bind-Value="sollicitant.Diploma">
                @foreach (var diploma in (Diploma[])Enum.GetValues(typeof(Diploma))) {
                    <div class="my-2 ms-3">
                        <InputRadio Value="diploma" class="form-check-input"
                            id="@diploma" />
                        <label class="form-check-label"
                            for="@diploma">@diplomas[(int)diploma]</label>
                     </div>
                }
            </InputRadioGroup>
        </div>
    </div>
    <div class="row my-3">
        <label for="functie" class="col-sm-2 col-form-label">Functie</label>
        <div class="col-sm-3">
            <InputSelect id="functie" class="form-control"
                         @bind-Value="sollicitant.Functie">
                <option value="">- Kies een functie -</option>
                <option>software tester</option>
                <option>junior .NET ontwikkelaar</option>
                <option>full stack .NET developer</option>
                <option>medior/senior developer</option>
            </InputSelect>
        </div>
    </div>
    <ValidationSummary />
    <button type="submit" class="btn btn-primary">Submit</button>
</EditForm>

@code {
    private static readonly List<string> diplomas = new List<string> { "Geen",
        "Middelbaar onderwijs", "Hoger onderwijs, niet universitair",
        "Hoger onderwijs, universitair" };
    private Sollicitant sollicitant = new Sollicitant();
    private void HandleValidSubmit() { }
}
```

15.9 JavaScript vanuit .NET oproepen

App.razor

```
    window.userInputFunctions = {
        …
        zetdefocus: function () {
            document.getElementById('getal').focus();
        }
    };
```

JSInterop.razor

```
protected async override Task OnAfterRenderAsync(bool firstRender)
{
    if (firstRender)
    {
        …
        await JSRuntime.InvokeVoidAsync(
            "userInputFunctions.zetdefocus");
    }
}
```

COLOFON

Domeinexpertisemanager	Bjorn Smeets
Moduleverantwoordelijke	Steven Lucas
Auteurs	Steven Lucas
Versie	28/03/2024

Omschrijving module-inhoud

Abstract	Doelgroep	.NET ontwikkelaar met C#
	Aanpak	Begeleide zelfstudie
	Doelstelling	Single Page Applications leren maken met ASP.NET Core Blazor
Trefwoorden		ASP.NET Core Blazor WebAssembly C# Single Page Applications
Bronnen/meer info		https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor https://learn.microsoft.com/nl-nl/aspnet/core/blazor