

Blazor

Framework SPA

Tables des matières

- Introduction
- Le pattern MVC
- Le pattern MVVM
- Web API Core
- SignalR

- Installation
- Premier projet

- Routing et Layout
- Components
- Les évènements
- Bindings
- Formulaires
- Virtualisation
- Consommation API
- JS Interop
- Sécurité
- Injection de dépendances

Introduction

Blazor

Introduction

- Le nom Blazor vient de Browser et Razor.
- Framework permettant de créer des SPAs(Single Page Applications) tout comme Angular, React, ViewJs... via C# et d'exécuter des librairies .NET standard dans notre navigateur.
- Ne requiert aucun plugin sur notre navigateur (contrairement au défunt Silverlight) et utilise le principe de WebAssembly supporté par la majorité des navigateurs.
- Il est open-source :
<https://github.com/dotnet/aspnetcore/tree/main/src/Components>
- Version Client & Server



Introduction

• ClientSide : WebAssembly(Wasm)

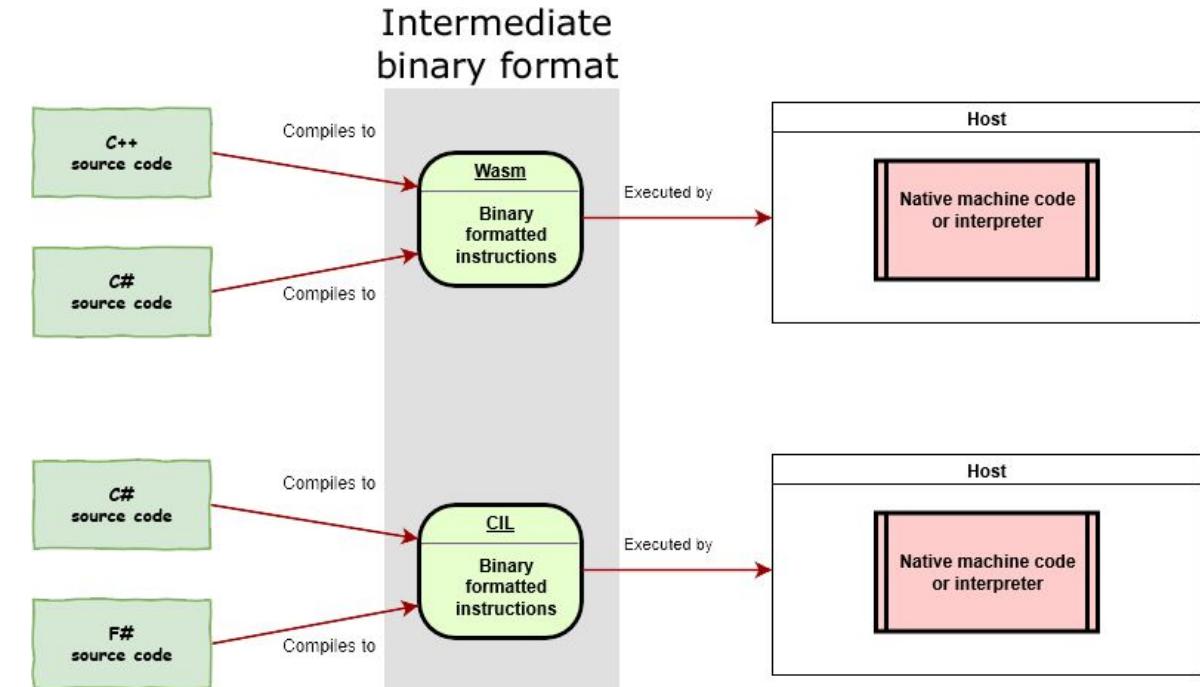
WebAssembly est un standard de binaire (byte code) compréhensible par les navigateurs modernes.

Le binaire WebAssembly est déjà compilé lorsqu'il arrive au navigateur :

- Il est 20 fois plus rapide qu'un fichier javascript
- Il est moins volumineux qu'un fichier js

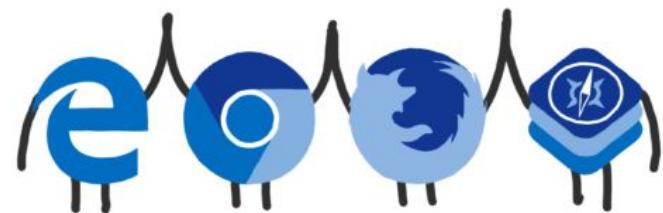
Il n'a pas pour but de remplacer Javascript mais plutôt pour compléter celui-ci avec des performances supérieures.

Exemple : [d3demo](#)



Introduction

	Your browser	Chrome ⁹¹	Firefox ⁸⁹	Safari ^{14.1}	Wasmtime ^{0.22}	Wasmer ^{2.0}
Standardized features						
JS BigInt to Wasm i64 integration	✓	✓	✓	✓	n/a	n/a
Bulk memory operations	✓	✓	✓	⌚	✓	✓
Multi-value	✓	✓	✓	✓	✓	✓
Import & export of mutable globals	✓	✓	✓	✓	✓	✓
Reference types	✗	⌚	✓	⌚	✓	✓
Non-trapping float-to-int conversions	✓	✓	✓	⌚	✓	✓
Sign-extension operations	✓	✓	✓	✓	✓	✓
Fixed-width SIMD	✓	✓	⌚	✗	⌚	✓
In-progress proposals						
Exception handling	✗	⌚	✗	✗	✗	✗
Module Linking	✗	✗	✗	✗	⌚	✗
Tail calls	✗	⌚	✗	✗	✗	✗
Threads and atomics	✓	✓	✓	⌚	✗	⌚

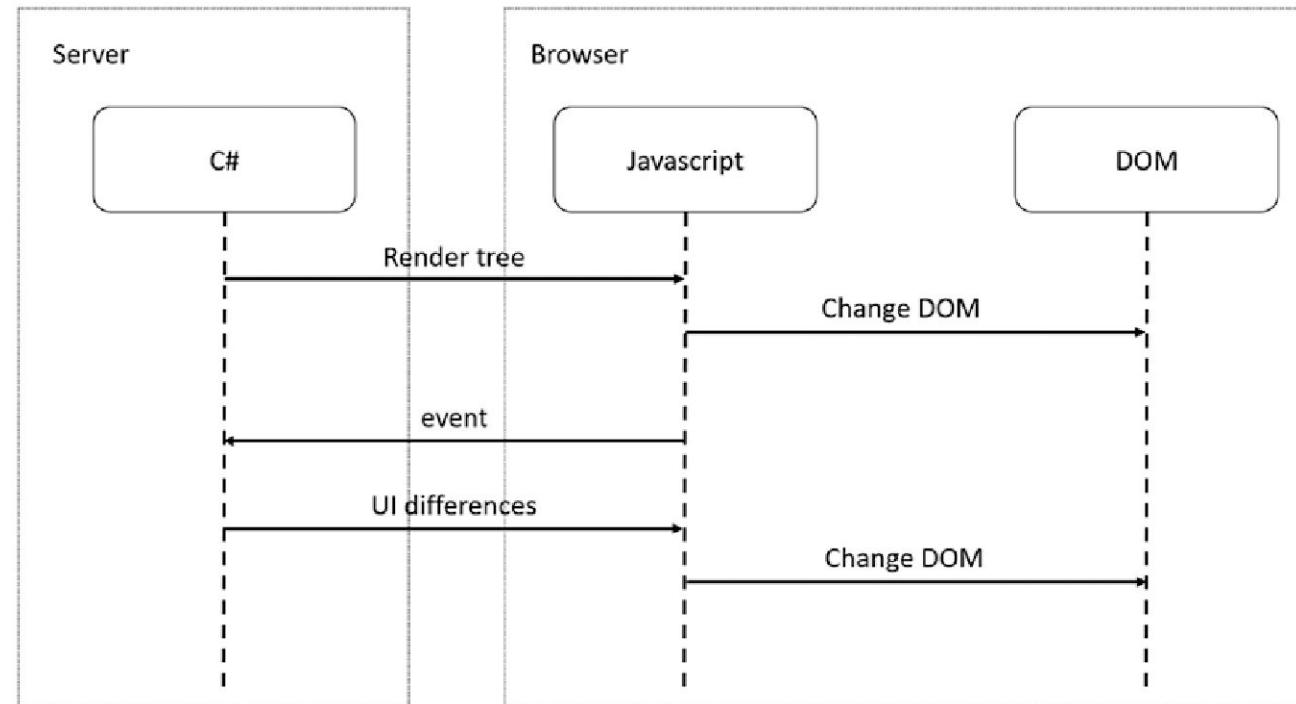


Introduction

Server-Side

Le principe est quasi identique au web assembly **SAUF** que c'est le serveur qui construit l'arbre DOM puis envoie la version sérialisé (js) au navigateur via SignalR.

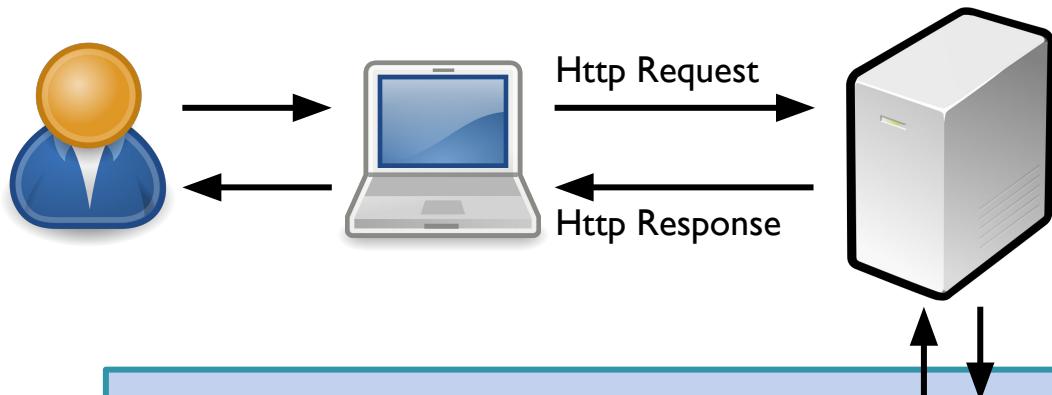
Javascript va alors désérialiser et mettre à jour le DOM du client et au besoin exécuter l'appel serveur pour les événements client.



Le pattern MVC

Model - View - Controller

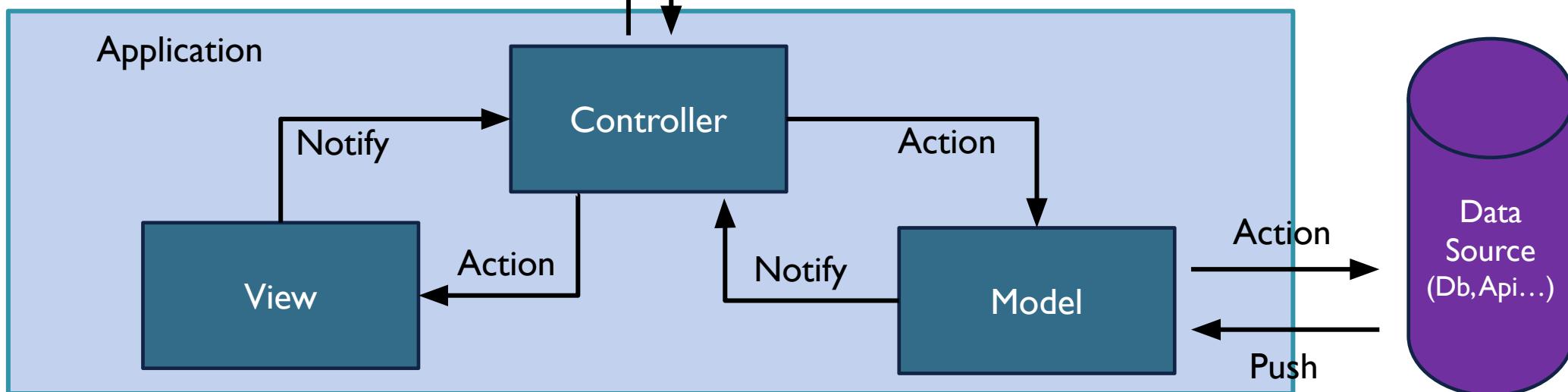
DESIGN PATTERN MVC



L'appel se fait avec une url structurée :
<https://www.contoso.com/product/details/5>

On y retrouve :

- le protocole (https)
- Le nom de domaine (contoso.com)
- L'entête (www)
- Les informations de routage (product/details/5)



DESIGN PATTERN MVC - RÔLES

Rôle du contrôleur

Dans le modèle MVC, le contrôleur est le point d'entrée initial de toutes requêtes faites par un client. Il est le garant de la logique métier de votre application et il est responsable de la sélection des types de modèle à utiliser et de la vue à afficher.

Ou en d'autres termes, ils gèrent l'interaction avec l'utilisateur, fonctionnent avec le modèle et, au final, si besoin sélectionnent une vue à afficher.

Rôle du modèle

Le modèle est la couche travaillant sur les données.

Ses rôles premiers sont de décrire les données et de fournir un accès à ces dernières (qu'il s'agisse d'une base de données, d'une api, ou d'un fichier plat).

Le cas échéant, le modèle définira les méthodes de conversions servant à transformer nos données avant l'envoie ou après réception afin que ces dernières appliquent les règles métier imposée par le corps business de l'applicatif.

Enfin, nous aurons parfois besoin de modèles de vue, ce type de modèle permettra l'affichage et/ou la gestion de formulaires (facilitant la validation de ces derniers).

DESIGN PATTERN MVC - RÔLES

Rôle de la vue

Les vues sont responsables de la présentation du contenu via l'interface utilisateur.

Elles utilisent le moteur d'affichage « Razor » pour incorporer le code .Net dans le balisage HTML.

Cependant, contrairement à d'autres Framework basés sur les fichiers, les vues ne sont pas directement accessibles mais fournissent un résultat à une demande spécifique du contrôleur.

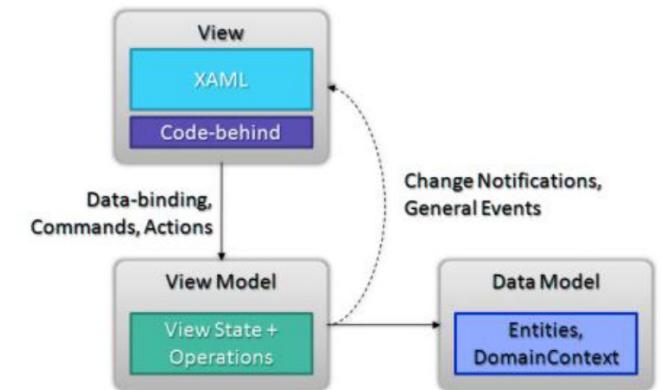
Le pattern MVVM

Model - View - ViewModel

HISTORIQUE

Le pattern MVVM a été évoqué, pour la première fois, en 2005 par John Gossman (architecte dans l'équipe de développement de WPF chez Microsoft Corporation) sur son blog, dans un message intitulé :

Introduction to Model/View/ViewModel pattern for building WPF apps
<http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx>



Au niveau des « fondations », John s'est appuyé sur deux patterns existants depuis très longtemps :

- Le pattern MVC (Model-View-Controller);
- Le pattern MVP (Model-View-Presenter).

Ensuite, il proposa d'utiliser les bases de certains patterns déjà existants et de les enrichir avec certaines fonctionnalités propres à WPF/Silverlight, à savoir :

- Le moteur de liaisons (*Binding*) ;
- Le système de commandes.

DESIGN PATTERN MVVM?

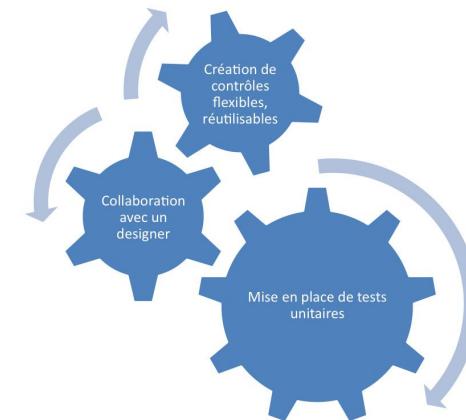
Un pattern est un ensemble de principes qui existent tous indépendamment les uns des autres mais qui, une fois réunis, forment un ensemble de règles qu'il est recommandé de respecter si l'on souhaite développer des applications maintenables, pérennes et compréhensibles par la majorité des développeurs.

Le pattern MVVM (Model-View-ViewModel), c'est tout simplement cela : un ensemble de techniques que l'on peut utiliser dans différents projets WPF, Silverlight et Windows Phone et qui permettent aux développeurs de disposer d'un code qui, au final, sera compris par tous les autres car il respecte des standards connus de tous.

Le gros avantage, c'est la souplesse de la maintenance et l'évolutivité qui en résultent, permettant à n'importe quel développeur de comprendre et faire évoluer le code.

D'autres avantages sont, également, à prendre en compte :

- Création de contrôles flexibles et réutilisable;
- Collaboration avec un designer;
- Mise en place de tests unitaires.



LES DIFFÉRENTS ÉLÉMENTS DU MVVM

Model :

Il correspond aux données utilisées par l'application et les moyens d'agir sur elles.

Il sera, par conséquent, la représentation technique des besoins fonctionnels sur les données exprimées par le commanditaire de l'application, comme par exemple : « un livre possède un titre et un nombre de pages », « je suis capable d'obtenir une liste de livres », « je suis capable d'ajouter un livre » ou encore « pour lire un livre, il faut qu'il existe ».

Les différents rôles et responsabilités de la partie modèle sont les suivants :

- définir le format des données ;
- définir les méthodes d'accès aux données (lecture, édition, suppression, création) ;
- assurer aux travers des services les différentes règles métier ;
- assurer éventuellement l'intégrité des données (validation) ;
- notifier le ViewModel des changements qui interviennent sur les données.

LES DIFFÉRENTS ÉLÉMENTS DU MVVM

View :

La vue est peut-être la partie la plus simple à comprendre car nous avons tous fait au moins une interface utilisateur. Dans le cadre de MVVM, nous aurons encore plus facile car notre vue ne sera pas couplée de façon forte avec les autres éléments et se contentera donc de remplir son rôle de représentation des données et des actions possibles provenant de notre « ViewModel ».

Elle remplit dans ce cadre trois rôles principaux :

- représenter tout ou une partie des données du « ViewModel » ;
- représenter les différents états du « ViewModel » ;
- notifier le « ViewModel » des différentes actions faites par l'utilisateur.

La vue est totalement libre d'utiliser le « ViewModel » comme bon lui semble et de choisir ce qu'elle souhaite représenter à l'écran ou non. Ceci est valable pour les données comme pour les actions (« Command »).

LES DIFFÉRENTS ÉLÉMENTS DU MVVM

View-Model :

Le ViewModel est l'élément central de l'architecture MVVM. Il est donc très important de connaître de façon précise quel est son rôle et sur quels principes il repose.

Pour reprendre la terminologie des langages objets que nous connaissons bien, le ViewModel peut être considéré comme l'interface – au sens programmatique du terme – de la vue. Il définit ce qui sera présent dans la vue, mais pas la façon dont cela sera implémenté.

Les principaux rôles du « ViewModel » sont :

- Exposer sous une forme bien précise les différentes entités à représenter : tris, filtres, compositions, sélection, extraction, etc.
- Exposer les différentes actions possibles sur ces entités.
- Gérer les différents états possibles de la vue (par exemple un mode édition et un mode lecture seule).
- Notifier la ou les vues rattachées des différents changements se produisant sur les éléments exposés.

ASP.NET CORE WEB API

Introduction

Introduction

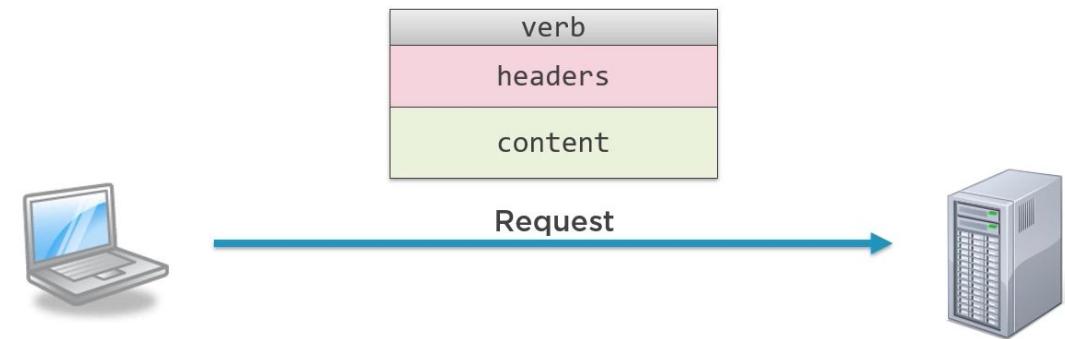
Avant de commencer à développer une Web API, il est important de bien comprendre certains concepts :

1. HTTP

Le protocole HTTP permet, basiquement, d'envoyer une *request* et de recevoir une *response* d'un serveur.

Le message transmis ou reçu est composé :

- Du verb : GET, POST, PUT, PATCH, DELETE, ...
- Du Headers : CONTENT_LENGTH, version de l'HTTP, mise en cache, ...
- Du Content : le contenu proprement dit du message



Introduction

2. Verbs

Les *Verbs* définissent les actions demandées au serveur et sont la base du système des Web API.

Les principaux Verbs sont :

- GET
Permet de **récupérer** une ressource au serveur (Pdf, Html, Js,)
- POST
Permet d'**ajouter** une nouvelle ressource au serveur
- PUT
Permet d' **écraser** une ressource existante (**mise à jour complète** de la ressource)
- PATCH
Permet une **mise à jour partielle** de la ressource sur le serveur
- DELETE
Permet la **suppression** d'une ressource sur le serveur

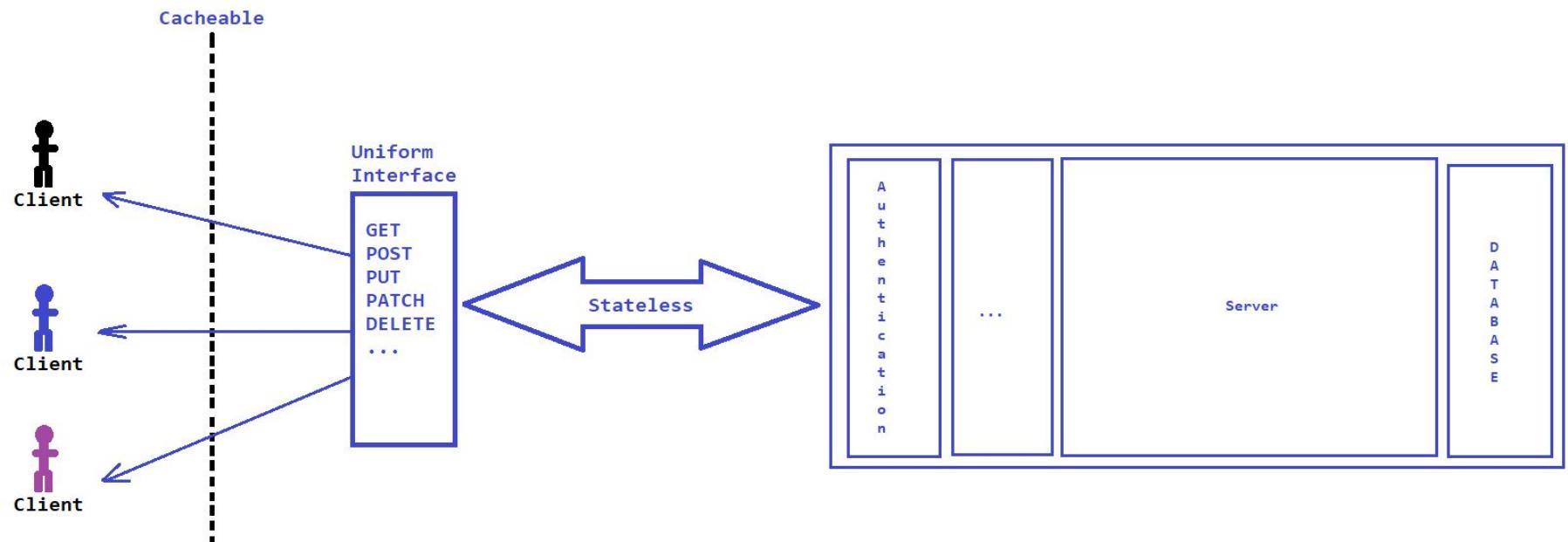
Introduction

2. REST

REpresentational State Transfer est un style d'architecture logicielle définissant un ensemble de contraintes à utiliser pour créer des services web.

Cette architecture inclus 5 contraintes :

- Uniform UI
- Client-Server
- Stateless
- Cacheable
- Layered System



Introduction

2.1 Uniform Interface

En appliquant le principe de généralité à l'interface des composants, nous pouvons simplifier l'architecture globale du système et améliorer la visibilité des interactions : Separation of concern, Modularity, Abstraction,...

Dans le cadre des API REST uniformes, les 4 contraintes suivantes doivent être respectées :

- **Identification des ressources** – L'interface doit identifier de manière unique chaque ressource impliquée dans l'interaction entre le client et le serveur.
- **Manipulation des ressources par des représentations** – Les ressources doivent avoir des représentations uniformes dans la réponse du serveur. Les consommateurs d'API doivent utiliser ces représentations pour modifier l'état des ressources sur le serveur.
- **Messages auto-descriptifs** – Chaque représentation de ressource doit contenir suffisamment d'informations pour décrire comment traiter le message. Il doit également fournir des informations sur les actions supplémentaires que le client peut effectuer sur la ressource.
- **Hypermédia comme moteur de l'état de l'application** – Le client ne doit avoir que l'URI initial de l'application. L'application cliente doit piloter dynamiquement toutes les autres ressources et interactions à l'aide d'hyperliens.

Introduction

2.2 Client-Server

Le modèle de conception client-serveur applique la **séparation des préoccupations (Separation of concerns)**, ce qui aide les composants client et serveur à évoluer indépendamment.

En séparant les problèmes d'interface utilisateur (client) des problèmes de stockage de données (serveur), nous améliorons la portabilité de l'interface utilisateur sur plusieurs plates-formes et améliorons l'évolutivité en simplifiant les composants du serveur.

Pendant que le client et le serveur évoluent, nous devons nous assurer que l'interface/le contrat entre le client et le serveur ne se rompt pas.

2.3 Stateless

- Exige que chaque demande du client au serveur contienne toutes les informations nécessaires pour comprendre et compléter la demande.
- Le serveur ne peut pas tirer parti des informations de contexte précédemment stockées sur le serveur.
- L'application cliente doit conserver entièrement l'état de session.

Introduction

2.4 Cacheable

La contrainte requiert qu'une réponse spécifie implicitement ou explicitement si elle peut être mise en cache ou non.

Si la réponse peut être mise en cache, l'application cliente obtient le droit de réutiliser les données de réponse ultérieurement pour des demandes équivalentes et une période spécifiée,

Nous devons donc utiliser le header HTTP pour renseigner les informations suivantes :

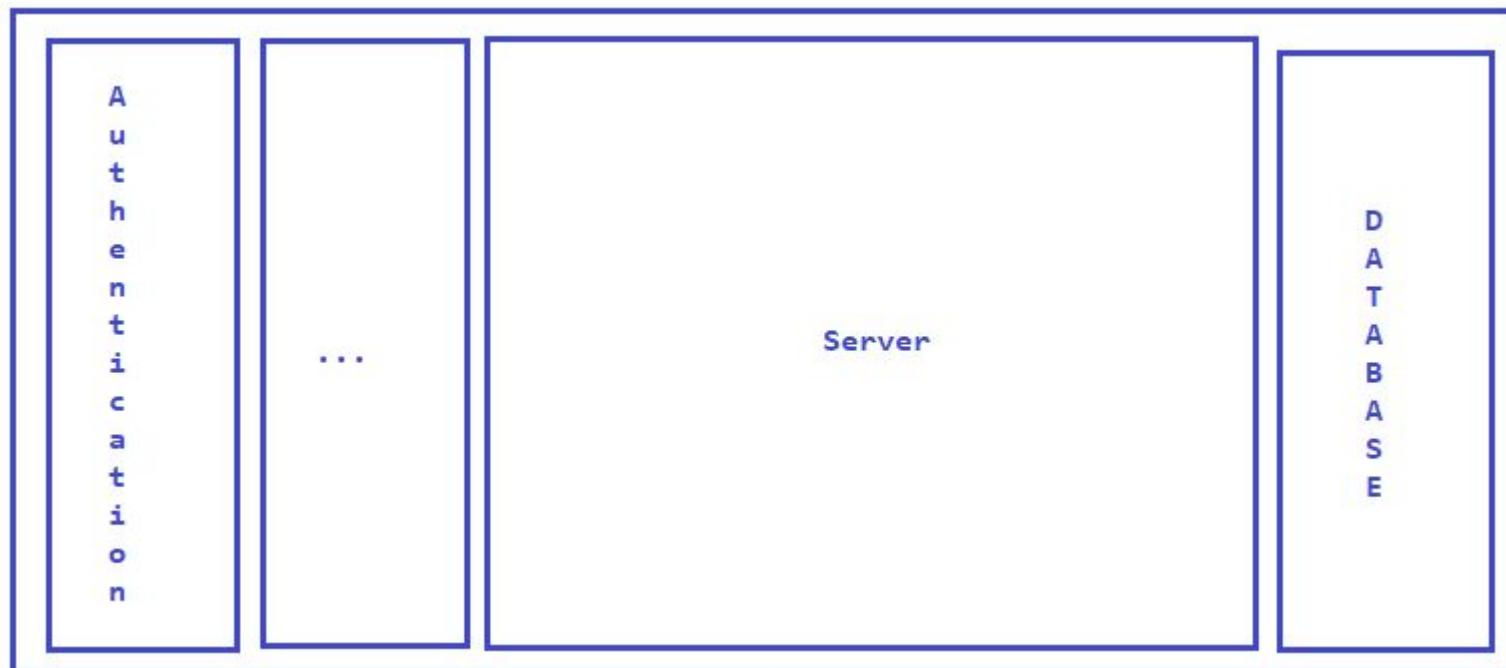
- **Expires HTTP header** : Expires: <http-date>
 - <https://developer.mozilla.org/fr/docs/Web/HTTP/Headers/Expires>
- **Cache-Control**
 - <https://developer.mozilla.org/fr/docs/Web/HTTP/Headers/Cache-Control>
- **Etag** : identifie la ressource de manière unique
 - <https://developer.mozilla.org/fr/docs/Web/HTTP/Headers/ETag>
- **Last-Modified** : Last-Modified: <nom-jour>, <jour> <mois> <année> <heure>:<minute>:<seconde> GMT
 - <https://developer.mozilla.org/fr/docs/Web/HTTP/Headers/Last-Modified>

Introduction

2.5 Modèle en couche

Le style de système en couches permet à une architecture d'être composée de couches hiérarchiques en contrignant le comportement des composants.

Par exemple, dans un système en couches, chaque composant ne peut pas voir au-delà de la couche immédiate avec laquelle il interagit.



SignalR

SignalR... C'est quoi?

SignalR est une bibliothèque pour ASP.Net qui permet de réaliser une communication bidirectionnelle en temps réel.

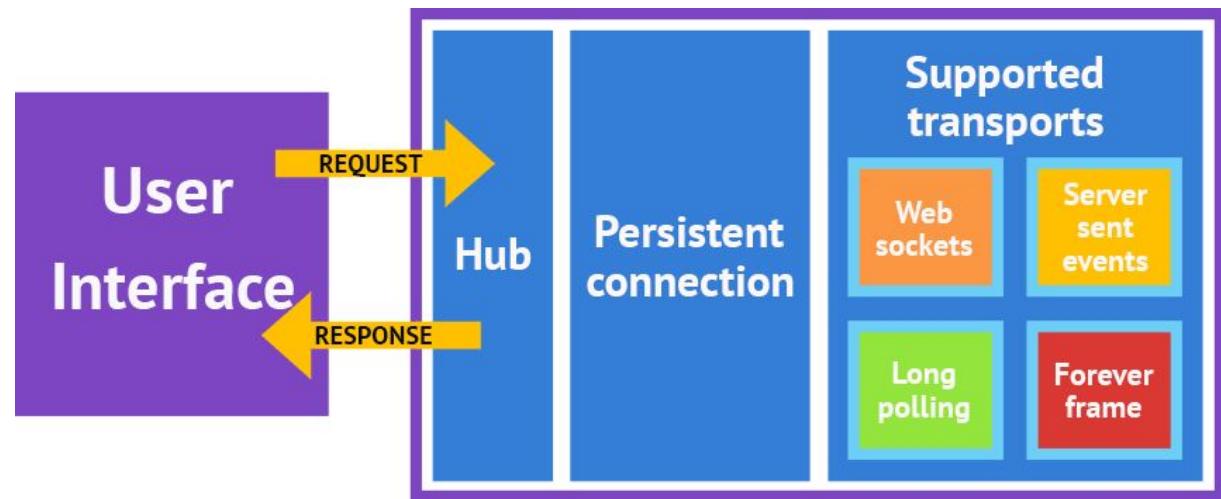
Son objectif est de permettre l'envois de contenu instantané entre client et serveur.

Pour cela, elle utilise :

- Une liaison “*continue*” avec les clients (WebSockets, Long polling, ...)
- Des classes « Hub » côté serveur, pour interagir avec les clients connectés.
- Des scripts Javascript, qui permet au client Web d’interagir avec le serveur.

Les protocoles

- WebSocket (HTML 5)
 - Protocole qui permet de créer un canal de communication full-duplex en TCP.
- Server Sent Events (HTML 5 - **Non compatible avec IE / Edge**)
 - API qui permet à un navigateur de recevoir une mise à jour à partir du serveur.
- Ajax long polling
 - Utilisation de requête ouverte en attente de réponse du serveur.
Lorsque la requête est terminée, une nouvelle requête est envoyée.



Quelle technologie utiliser ?

- Si JSONP est configuré à true → **Ajax long polling**
- Si infrastructure cross-domain (hébergement différent entre la page et signalR)
 - a. Utilisation de **WebSockets**, si le client et le serveur le support.
 - b. Sinon utilisation d'**Ajax long polling**
- Dans les autres cas, en fonction de la compatibilité du client et du serveur
 - a. Utilisation de **WebSocket**, si le client et le serveur le support.
 - b. Sinon, **Server Sent Events** si l'option est disponible
 - c. Sinon utilisation d'**Ajax long polling**

Communication

SignalR fournit deux modes de communications :

1) Persistent Connections (Net Framework)

Les connexions persistantes fournissent un accès direct à un protocole de communication de bas niveau fourni par signalR. Chaque connexion client est identifiée par un ID de connexion.

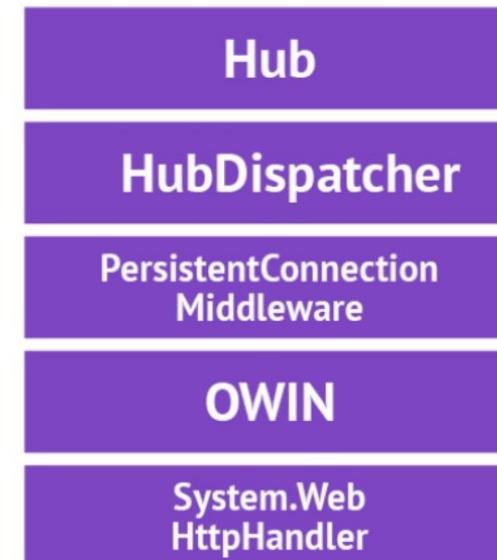
2) Hubs (Net framework & Net Core)

Les hubs fournissent une API de haut niveau permettant au client et au serveur d'appeler leurs méthodes respectives. Ce modèle est facile à mettre en œuvre par les développeurs habitués à appeler des API distantes

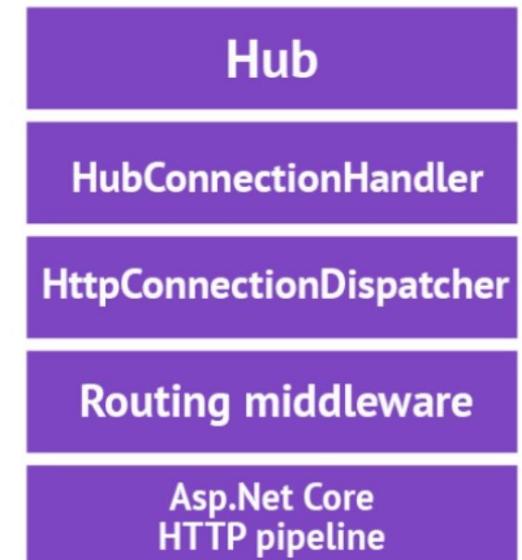
Protocole:

Asp.net Core supporte le JSON et un protocole binaire basé sur MessagePack

Original SignalR



Asp.Net Core SignalR



Communication

MessagePack est un format de sérialisation binaire rapide et compact. C'est utile lorsque les performances et la bande passante sont un problème, car cela crée des messages plus petits par rapport à JSON.

Les messages binaires sont illisibles lorsque vous consultez les traces et les journaux du réseau, sauf si les octets sont passés par un analyseur MessagePack.

SignalR a une prise en charge intégrée du format MessagePack et fournit des API que le client et le serveur peuvent utiliser

Introduction

Outre ses règles de base, Microsoft de son côté, propose ces deux liens pour les principes de design des Api

- <https://github.com/Microsoft/api-guidelines/blob/master/Guidelines.md>
- <https://mathieu.fenniak.net/the-api-checklist/>

Installation

Blazor

Installation

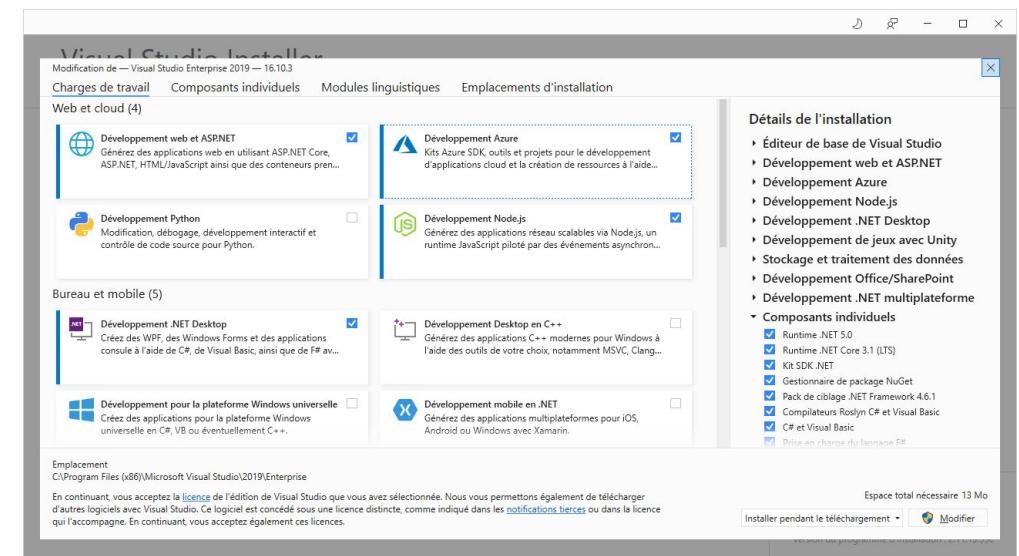
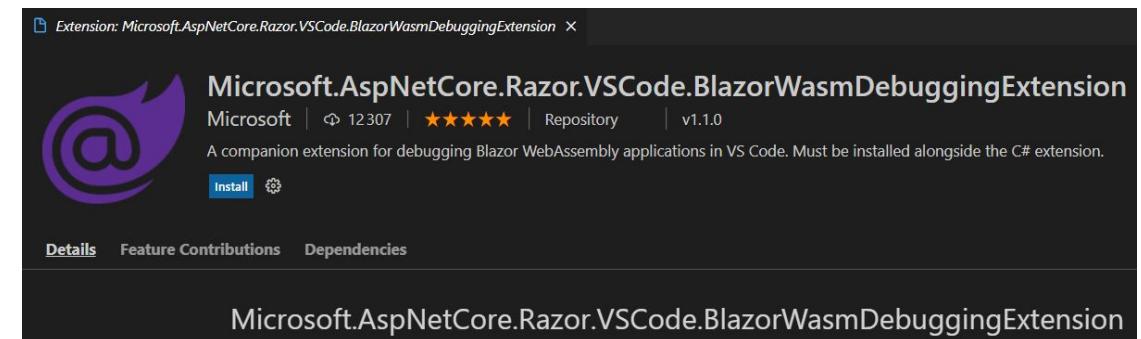
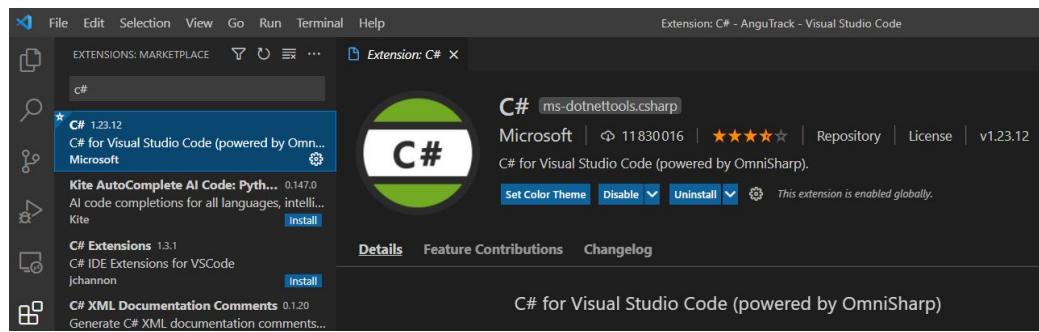
Télécharger Visual Studio 2019

(<https://visualstudio.microsoft.com/fr/vs/community/>)

Et ensuite, lors de l'installation, dans le choix des composants, vous devez choisir *Développement web et ASP.NET*.

Il est également possible de développer via Visual studio code
(https://code.visualstudio.com/?wt.mc_id=DX_841432)

Et ensuite vous pourrez installer l'extension C# et BlazorWasmDebuggingExtension



Installation

Vous pourriez avoir également besoin des templates Blazor pour VSCode pour créer vos projets en ligne de commande.

Pour cela, exécutez la commande suivante dans un terminal/invite de commande/Powershell

```
dotnet new -i Microsoft.AspNetCore.Components.WebAssembly.Templates
```

```
PS D:\Cours\Blazor\Exemples> dotnet new -i Microsoft.AspNetCore.Components.WebAssembly.Templates
Bienvenue dans .NET 5.0 !
-----
Version du kit SDK : 5.0.301

Télémétrie
-----
Les outils .NET collectent des données d'utilisation qui nous aident à améliorer votre expérience utilisateur. Elles
r la communauté. Vous pouvez refuser l'adhésion à la téléémétrie en affectant la valeur '1' ou 'true' à la variable d'environnement DOTNET_CLI_TELEMETRY_OPT_IN. Pour plus d'informations sur la téléémétrie des outils CLI .NET, accédez à https://aka.ms/dotnet-cli-telemetry

-----
Un certificat de développement HTTPS ASP.NET Core a été installé.
Pour approuver le certificat, exécutez 'dotnet dev-certs https --trust' (Windows et macOS uniquement).
Découvrez HTTPS : https://aka.ms/dotnet-https

-----
Écrivez votre première application : https://aka.ms/dotnet-hello-world
Découvrez les nouveautés : https://aka.ms/dotnet-whats-new
Consultez la documentation : https://aka.ms/dotnet-docs
Signalez des problèmes et recherchez du code source sur GitHub : https://github.com/dotnet/core
Utilisez 'dotnet --help' pour voir les commandes disponibles, ou accédez à https://aka.ms/dotnet-cli

-----
Préparation en cours... Merci de patienter.



| Nom du modèle       | Nom court | Langue       | Balises        |
|---------------------|-----------|--------------|----------------|
| Console Application | console   | [C#], F#, VB | Common/Console |
| Class library       | classlib  | [C#], F#, VB | Common/Library |
| WPF Application     | wpf       | [C#], VB     | Common/WPF     |
| WPF Class library   | wpflib    | [C#], VB     | Common/WPF     |


```

Premier Projet

Blazor

Premier Projet

Nous avons plusieurs choix.

- Un projet standalone (template Blazorwasm):
Ne nécessite pas de code côté serveur. Simple à déployer puisqu'un simple transfert des fichiers suffit à ce que notre navigateur l'interprète comme un site web classique
- Un projet hébergé :
Un client + 1 server + Du code partagé. Nécessite d'avoir le framework core installé pour notre hébergement car une partie du code sera managé côté serveur
- Un projet pure Server :
Le browser utilise SignalR pour recevoir les mises à jour de l'UI et envoyé les événements

Nous choisirons pour notre approche la seconde solution : WebAssembly Blazor hébergé dans une application ASP.NET Core.

Cela nous permettra de nous familiariser avec Blazor en utilisant la puissance du Debug vs car le Debugging pure des WebAssembly est limitée.

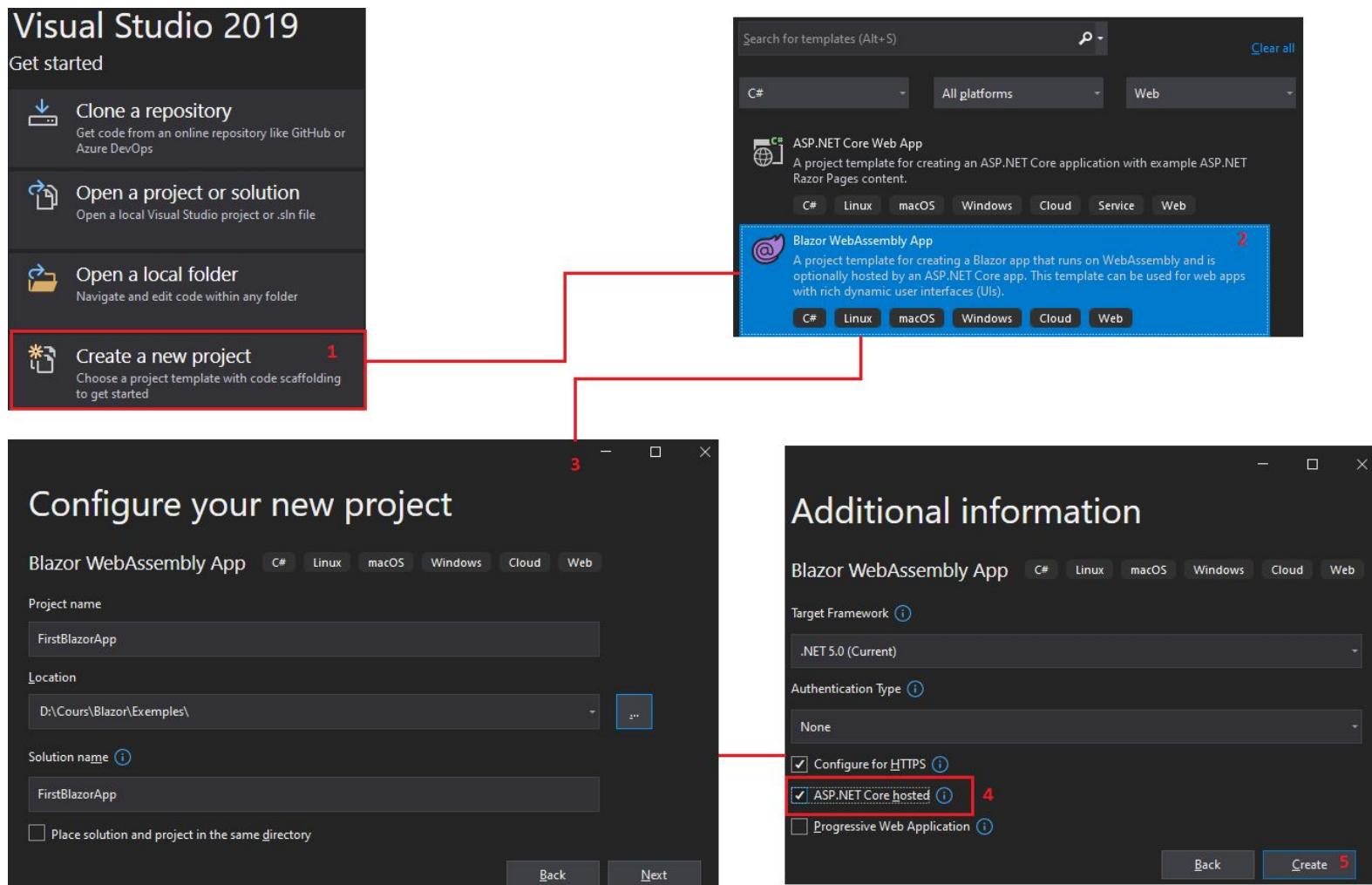
Cependant, tous les concepts vus pourront ensuite être appliqués aux deux autres choix .

Premier Projet

1. Ouvrir Visual Studio
2. Créer un nouveau projet
3. Sélectionner *Blazor WebAssembly App*
4. Entrer un nom de projet
5. Checker la box *Asp.NET Core Hosted*
6. Cliquer sur *Create*

Vous pouvez également utiliser la ligne de commande suivante :

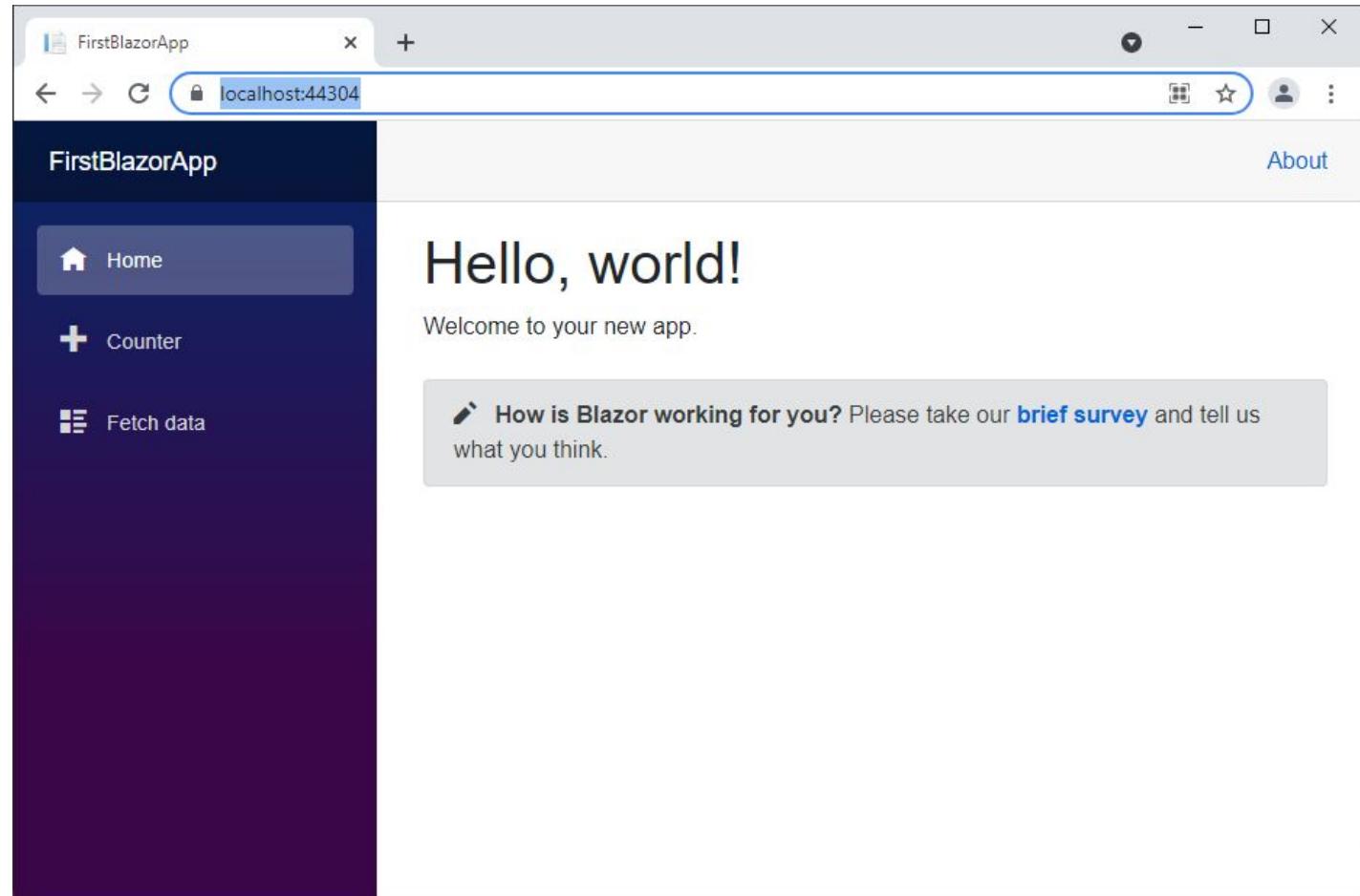
```
dotnet new blazorwasm --hosted -o FirstBlazorAppCli
```



Premier Projet

Un projet « exemple » est créé et vous pouvez lancer directement celui-ci pour voir votre première app.

(Via F5 sur Visual Studio ou via la ligne de commande `dotnet run` (dans le dossier contenant la partie server))



Structure de la solution

Notre solution est décomposée en 3 projets :

1. > Server

« Simple » application ASP.NET Core configurée pour permettre l'utilisation de blazor et exposant également un endpoint (Weather) pour permettre l'envoi des données vers les pages razor.

Le fichier de configuration (startup.cs) configure l'application pour permettre l'utilisation de Blazor

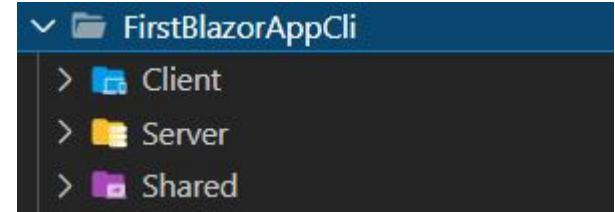
```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseWebAssemblyDebugging();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-h
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseBlazorFrameworkFiles(); ==> Configure l'application pour servir les fichiers de l'infrastructure
    app.UseStaticFiles();           webassembly Blazor à partir du chemin d'accès racine « / »

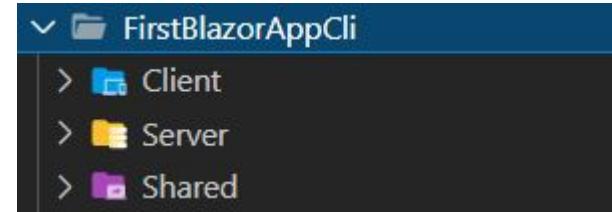
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
        endpoints.MapControllers();
        endpoints.MapFallbackToFile("index.html"); ==> Permet de "router" les appels non lié à l'API (pas un nom de fichier
                                                       ,ni une route qui match avec les routes définies)
    });
}

!!!Toujours en dernier dans notre config!!!
```



Structure de la solution

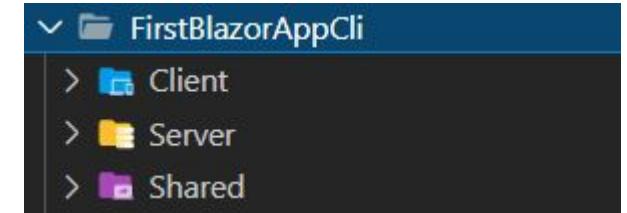


2. > Shared

Ce projet contient les « Models » qui vont être utilisés pour envoyer les données aux vues blazor

The screenshot shows the Visual Studio IDE with the 'WeatherForecast.cs' file open in the code editor. The code defines a class 'WeatherForecast' with properties for Date, TemperatureC, Summary, and TemperatureF.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace FirstBlazorAppCli.Shared
6  {
7      public class WeatherForecast
8      {
9          public DateTime Date { get; set; }
10
11         public int TemperatureC { get; set; }
12
13         public string Summary { get; set; }
14
15         public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
16     }
17 }
18 }
```



Structure de la solution

3. > Client

Contient la partie Client destinée à exposer nos composants Blazor.

Si nous ouvrons le fichier Index.html (dans wwwroot), nous remarquons une div ayant l'id app : permet l'insertion d'un composant Blazor.

```
<div id="app">Loading...</div>
```

Nous avons également un script associé à notre page :

```
<script src="_framework/blazor.webassembly.js"></script>
```

Ce script installe Blazor en téléchargeant *dotnet.wasm* et nos assemblies

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=no" />
    <title>FirstBlazorAppCli</title>
    <base href="/" />
    <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
    <link href="css/app.css" rel="stylesheet" />
    <link href="FirstBlazorAppCli.Client.styles.css" rel="stylesheet" />
</head>

<body>
    <div id="app">Loading...</div>

    <div id="blazor-error-ui">
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">X</a>
    </div>
    <script src="_framework/blazor.webassembly.js"></script>
</body>

</html>
```

Layout

Pour que `<div id="app">Loading...</div>` puisse recevoir le composant principal nous devons regarder du côté du Program.cs :

```
public class Program
{
    0 references
    public static async Task Main(string[] args)
    {
        var builder = WebAssemblyHostBuilder.CreateDefault(args);
        builder.RootComponents.Add<App>("#app");

        builder.Services.AddScoped(sp => new HttpClient
        { BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) });

        await builder.Build().RunAsync();
    }
}
```

Routing & Layout

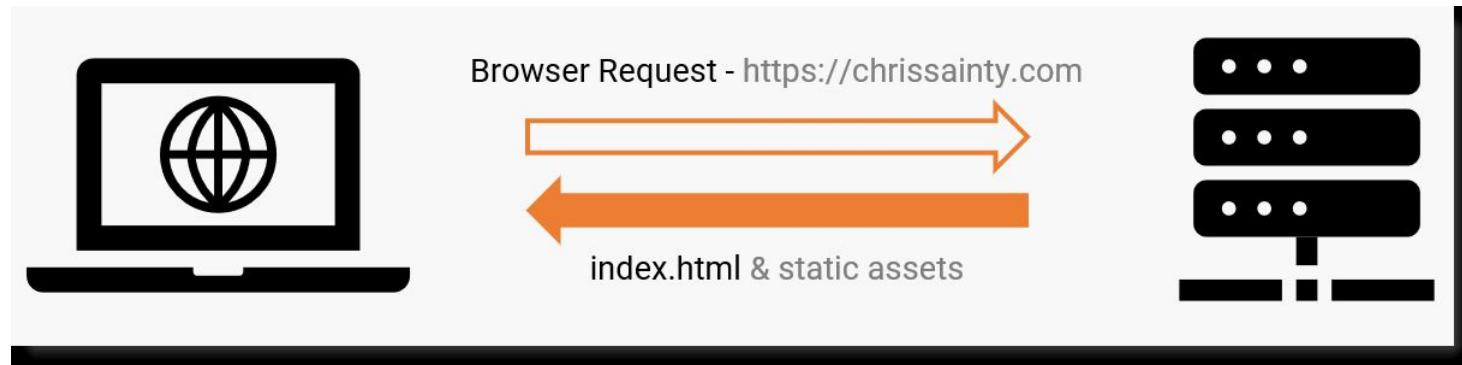
Blazor

Routing

Blazor prend en charge le routage d'URL.

Le routage d'URL vous permet de configurer une application pour accepter les URL de demande qui ne correspondent pas à des fichiers physiques mais utilisent une route sémantiquement significative.

Lors du chargement initial d'un site SPA ce n'est pas très différent des applications Web traditionnelles: Nous effectuons la requête HTTP GET, puis téléchargeons le HTML, le CSS, le JavaScript et tout autre élément statique.



Routing

La différence vient lors des changements de pages par la suite :



Cette fois, la SPA ne charge que les données du serveur.

En effet, les applications SPA ont tendance à télécharger l'intégralité de l'application lors du premier chargement du site, donc tout est déjà là. Lors du changement entre les pages, le seul contenu supplémentaire requis est les données à afficher.

Routing

Router

Lors de la création du projet Blazor, le *Router* est automatiquement installé et se trouve dans *App.razor*.

```
<Router AppAssembly="@typeof(Program).Assembly" PreferExactMatches="@true">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

Routing

Deux templates sont présent dans notre *App.razor* :

- Le premier permet de renvoyer la vue existante suivant la route

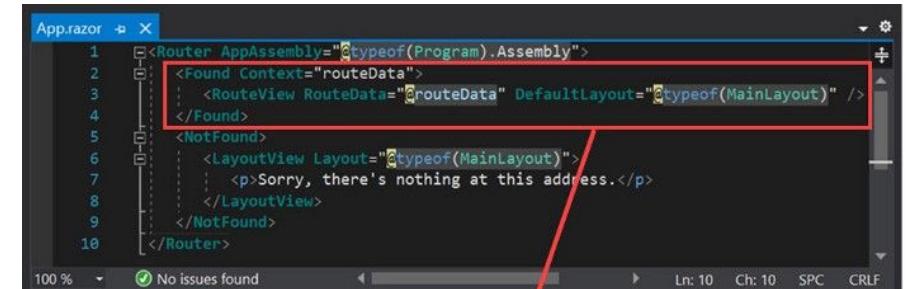
```
<Found Context="routeData">
    <RouteView
        RouteData="@routeData"
        DefaultLayout="@typeof(MainLayout)" />
</Found>
```

- Le second permet de renvoyer un message si la route n'existe pas

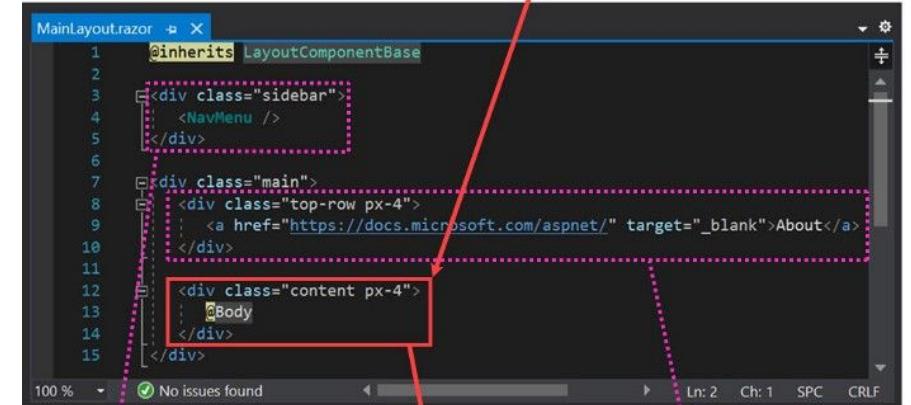
```
<NotFound>
    <LayoutView Layout="@typeof(MainLayout)">
        <p>Sorry, there's nothing at this address.</p>
    </LayoutView>
</NotFound>
```

Ces 2 templates utilisent un habillage commun : un layout.

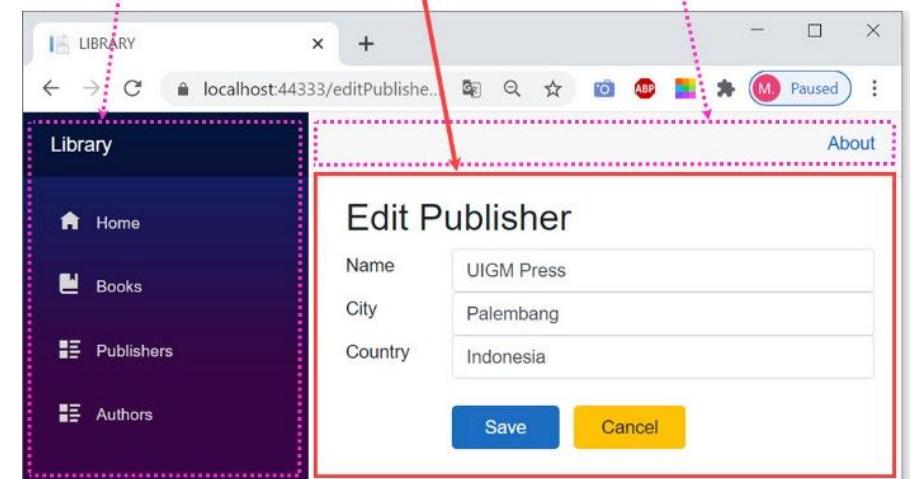
Celui-ci, *MainLayout*, définit le visuel commun des pages et permet l'injection du contenu du composant grâce à l'instruction *@Body*



```
App.razor
1 <Router AppAssembly="@typeof(Program).Assembly">
2     <Found Context="routeData">
3         <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
4     </Found>
5     <NotFound>
6         <LayoutView Layout="@typeof(MainLayout)">
7             <p>Sorry, there's nothing at this address.</p>
8         </LayoutView>
9     </NotFound>
10    [</Router>]
```



```
MainLayout.razor
1 @inherits LayoutComponentBase
2
3     <div class="sidebar">
4         <NavMenu />
5     </div>
6
7     <div class="main">
8         <div class="top-row px-4">
9             <a href="https://docs.microsoft.com/aspnet/" target="_blank">About</a>
10        </div>
11
12        <div class="content px-4">
13            @Body
14        </div>
15    </div>
```



Routing

- Définition des routes

Les routes doivent être définies sur chaque page via l'instruction @page.

Exemple :

Index.razor

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.
```

Counter.razor

```
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>
```

Routing

Nous pouvons également les définir en « code-behind »(voir chapitre components) au dessus de la classe.

Il est également possible de définir plusieurs routes pour un même composant

```
@page "/contact"
@page "/ContactMe"


<div class="top-header">
        <div class="logo">


```

Remarque :

Péférrez la gestion des routes dans les pages plutôt que dans le code-behind

```
[Route("/Contact")]
3 references
public class ContactBase : ComponentBase
{
    private string _title, _message;

    2 references
    public string Title...
    2 references
    public string Message...

    0 references
    public ContactBase()
    {
        Title = "Contactez-nous";
        Message = "Bienvenue";
    }
}
```

Routing

- Paramètres

Il est possible de transmettre des paramètres via les routes afin de personnaliser l'affichage...

Nous aurons alors une déclaration de route comme suit : `@page "/edit/{nom:type}"`

Le type par défaut d'un paramètre est `string` mais nous pouvons bien entendu

définir d'autres types pour nos paramètres de route :

Type	Exemple
bool	{isActive:bool}
datetime	{dob:datetime}
decimal	{gpa:decimal}
double	{height:double}
float	{weight:float}
guid	{id:guid}
int	{id:int}
long	{isbn:long}

Routing

```
@page "/counter/{deb:int?}"  
  
<h1>Counter - @Deb</h1>  
  
<p>Current count: @currentCount</p>  
  
<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>  
  
@code [  
    private int currentCount = 0;  
    @code{  
        [Parameter]  
        public int? Deb { get; set; }  
  
    }  
    public Counter()  
    {  
        if (Deb.HasValue) currentCount = Deb.Value;  
    }  
    private void IncrementCount()  
    {  
        currentCount++;  
    }  
]
```

Navigation Manager

Blazor

Navigation Manager

Le service NavigationManager nous permet de gérer les URI et la navigation en code C#.

Cette classe possède les properties suivantes :

BaseUrl	Le <i>BaseUrl</i> est toujours représenté comme un URI absolu sous forme de chaîne avec une barre oblique à la fin.(Correspond à l'attribut 'href' sur l'élément <base> du document)
Uri	Représente l'Uri courante sous la forme d'une chaîne de caractère (Uri absolue)

```
1 @page "/"
2 @inject NavigationManager nvm
3
4 <div class="text-center">
5   <p>URI : @nvm.Uri</p>
6   <p>Base Uri : @nvm.BaseUri</p>
7 </div>
```



URI : https://localhost:44346/

Base Uri : https://localhost:44346/

Navigation Manager

Et les méthodes/Events suivants :

NavigateTo	Permet de naviguer vers l'url spécifié
ToAbsoluteUri	Convertit une Uri relative en Absolue
ToBaseRelativePath	Convertit une Uri absolue vers une Uri relative au BaseUri.
LocationChanged	Événement lancé lors de changement de page

```
3   <button class="btn btn-primary" @onclick="GoToHome">
4     Home Page
5   </button>
6   <button class="btn btn-primary" @onclick="GoToCounter">
7     Counter Page
8   </button>
9
10  @code {
11    private void GoToHome()
12    {
13      nvm.NavigateTo("/");
14    }
15
16    private void GoToCounter()
17    {
18      nvm.NavigateTo("counter");
19    }
20 }
```

Routing

Nous ne sommes pas obligés de gérer la navigation en c#.

Il existe un composant Blazor : NavLink.

```
<NavLink class="nav-link" href="" Match="NavLinkMatch.All">
|   <i class="home"></i>
</NavLink>
```

Nous avons le choix entre :

1. NavLinkMatch.All : La navigation se fera si ça l'url entière match
2. NavLinkMatch.Prefix: La navigation se fera lorsque l'url match avec le préfix de l'url.

Layout

Blazor

Layouts

En règle générale, les applications Web modernes contiennent plus d'une page avec certains éléments de mise en page tels que des menus, des logos, des messages de copyright, etc. qui sont présents sur toutes les pages.

Pour faciliter la maintenance, il est impensable de « copier-coller » les zones communes sur chaque page...

Le concept de Layout vient donc résoudre ce problème.

Pour qu'un contenu soit un Layout , il doit :

- hériter de *LayoutComponentBase*
- *Avoir une instruction @Body permettant d'injecter le contenu des pages dépendante de ce layout*

```
@inherits LayoutComponentBase



<header>
        |   <h1>This is the header</h1>
    </header>

    <div class="content">
        |   @Body
    </div>

    <footer>
        |   This is the footer
    </footer>


```

Layout

Une fois notre Layout créé, la façon la plus simple de l'utiliser est de spécifier l'instruction `@layout nomLayout` au dessus de nos pages.

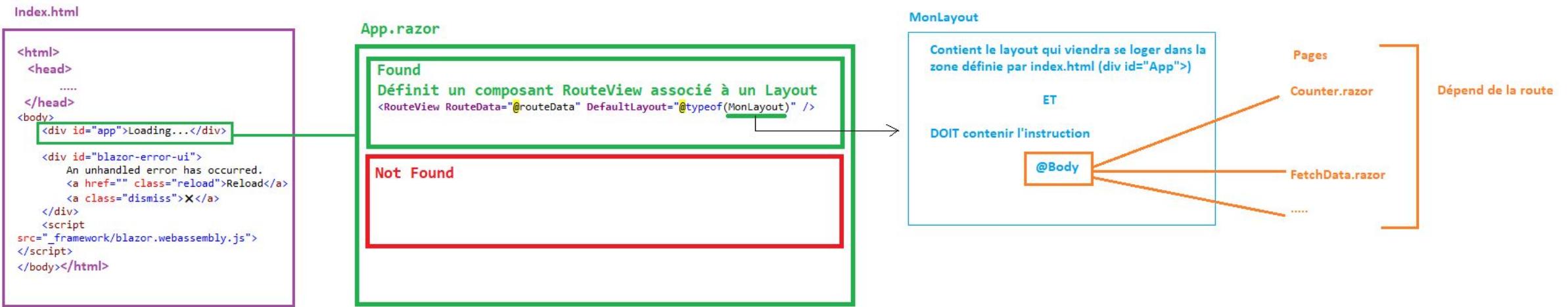
```
@page "/"
@layout MainLayout
<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />
```

Layout

Structure générale



Components

Blazor

Components

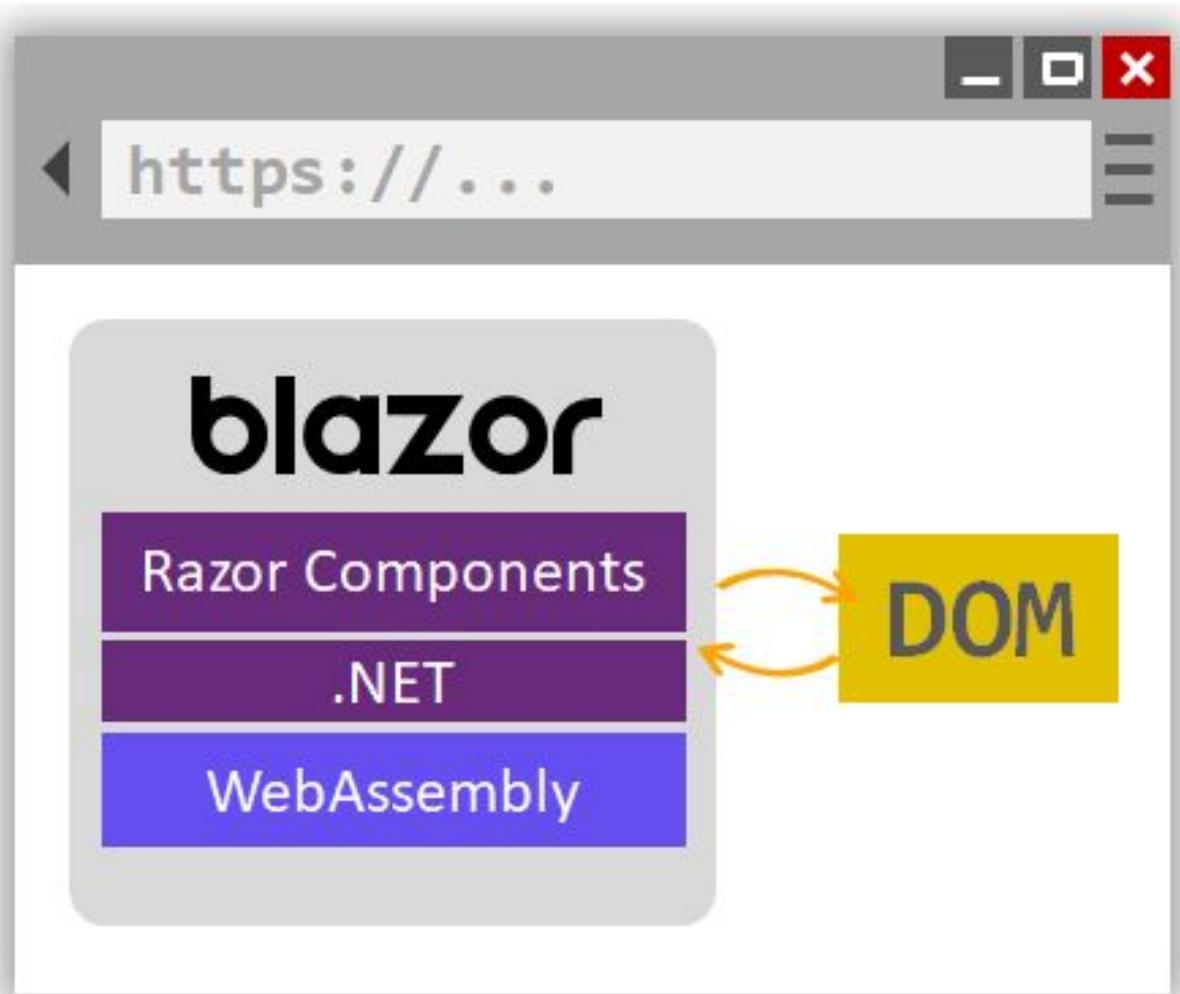
Un composant est une partie autonome de l'interface utilisateur (UI), telle qu'une page, une boîte de dialogue ou un formulaire.

Le but de ces composants est de les imbriquer, réutiliser et les partager entre les projets.

Toutes les pages sous Blazor sont considérées comme des composants.

Un composant est la combinaison de :

- HTML (UI)
- C# (Logique)



Components

Anatomie d'un composant

Trois approches existent pour la création d'un composant :

- 1) Single File
Le code HTML et c# se retrouvent dans le même fichier Razor (Default)
- 2) Partial File
Le code c# est placé dans un fichier portant le même nom que notre fichier razor sous la forme d'une classe partielle
- 3) Base Class
Le code c# est placé dans une classe héritant de *ComponentBase*. Ensuite nous ajoutons l'instruction @inherit pour faire hériter notre page razor de notre classe de base

La première approche est à déconseiller car elle ne sépare pas correctement le visuel et la logique.

Entre les deux autres approches, le choix dépendra de nos intentions :

- Si nous désirons simplement séparer mais ne pas utiliser l'encapsulation pour les accès aux propriétés, ni gérer le cycle de vie du composant : PartialFile
- Sinon : Base Class

Components

Single File

Un seul Fichier pour l'HTML et le C#.

- Avantage :

Aucun

- Désavantages :

Maintenabilité

Testabilité

Réutilisation

```
// Counter.razor

@page "/counter"

<h1>Counter</h1>

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

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

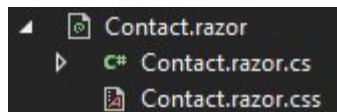
@code {
    private int currentCount = 0;

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

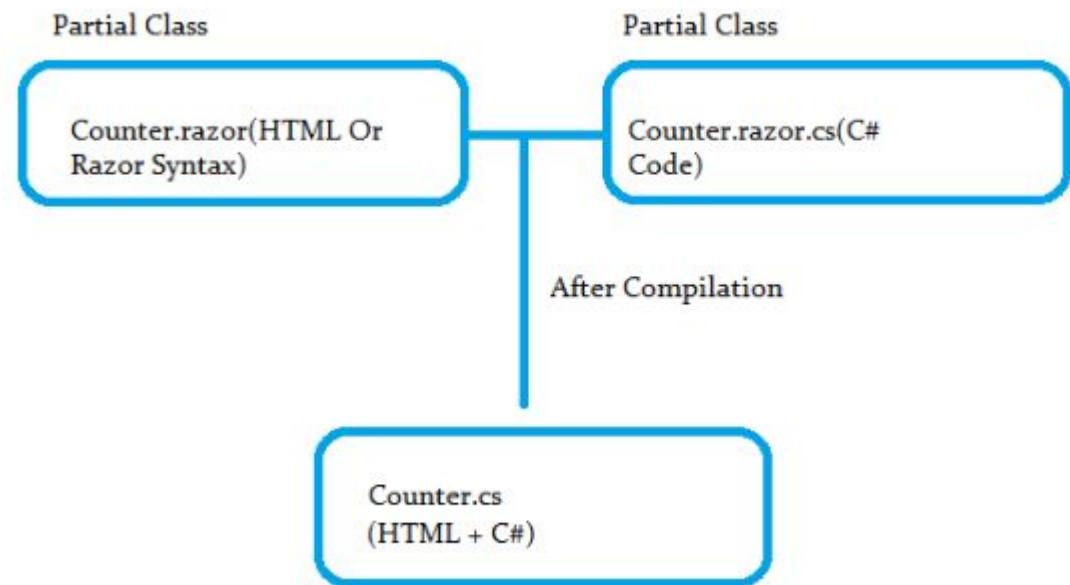
Components

Partial Class

Nous créons un fichier contenant une partial classe portant le nom de notre composant.



- Avantage :
 - Séparation des layers
 - Maintenabilité
 - Testabilité
- Désavantages :
 - Gestion minimale du cycle de vie du component



Components

```
@page "/counter/{deb:int?}"  
  
<h1>Counter - @Deb</h1>  
  
<p>Current count: @currentCount</p>  
  
<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
```

```
public partial class Counter  
{  
    private int currentCount = 0;  
  
    [Parameter]  
    3 references  
    public int? Deb { get; set; }  
  
    0 references  
    public Counter()  
    {  
        if (Deb.HasValue) currentCount = Deb.Value;  
    }  
    1 reference  
    private void IncrementCount()  
    {  
        currentCount++;  
    }  
}
```

Components

Base Class

Nous créons une classé héritant de ComponantBase et nous faisons hérité notre Razor page de cette classe

Avantage :

Séparation des layers

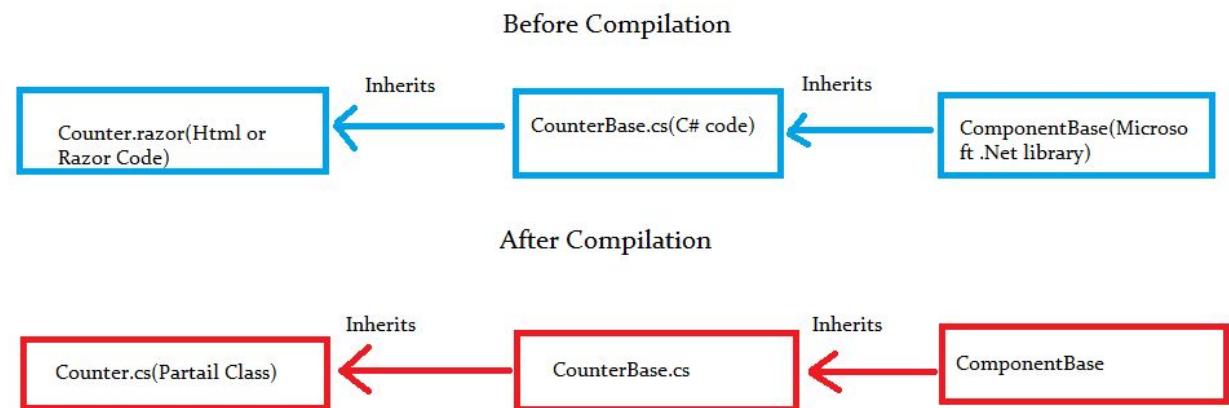
Testabilité

Maintenabilité

Gestion facilitée du cycle de vie du composant

- Désavantages :

Demande un peu de temps supplémentaire pour l'implémentation



Component

Les événements

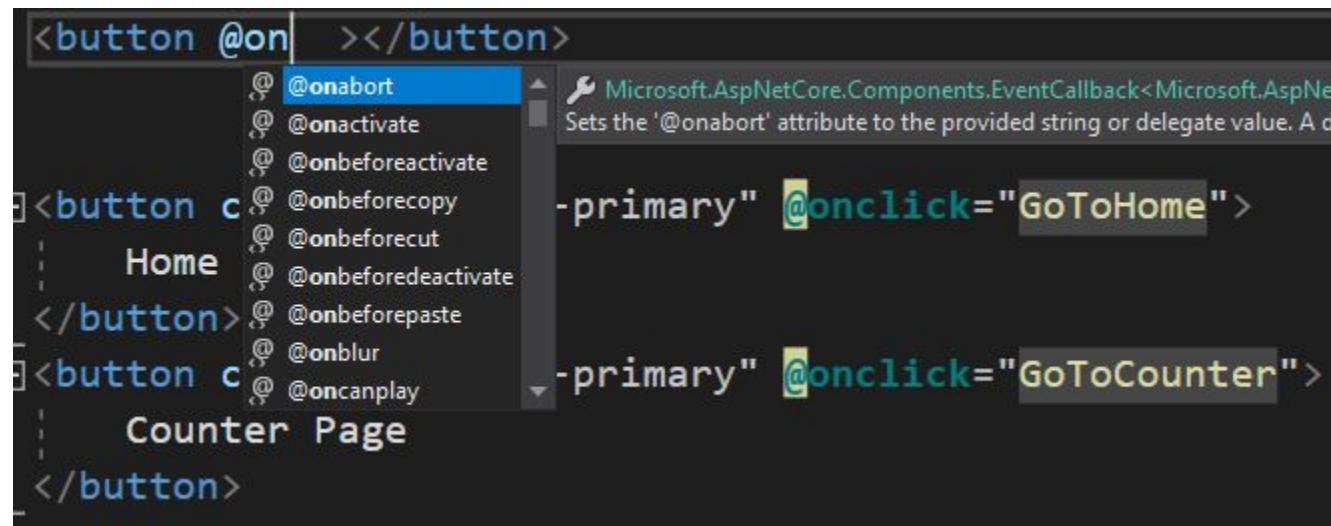
Components

Dom Events

Les événements

Dom Events

Les interactions utilisateur via l'Html sont gérés par des directives Blazor commençant par @on.



Ces directives prennent un délégué comme paramètre..

Remarque :

Lors de l'écriture d'une application Blazor, Blazor accroche les événements dans le navigateur et les envoie au serveur afin que les méthodes C# puissent être exécutées. Cela peut entraîner un ralentissement notable pour les événements fréquemment déclenchés tels que onmousemove.

EventCallback<T>

Les événements

Les événements

Il est possible de créer des événements dans nos composants basés sur la classe *EventCallback<T>*.

- Ajout d'un event

Il suffit de déclarer un paramètre public de type EventCallback

```
[Parameter]
1 reference
public EventCallback<int> OnMultipleOfThree { get; set; }
```

- Abonnement à l'événement

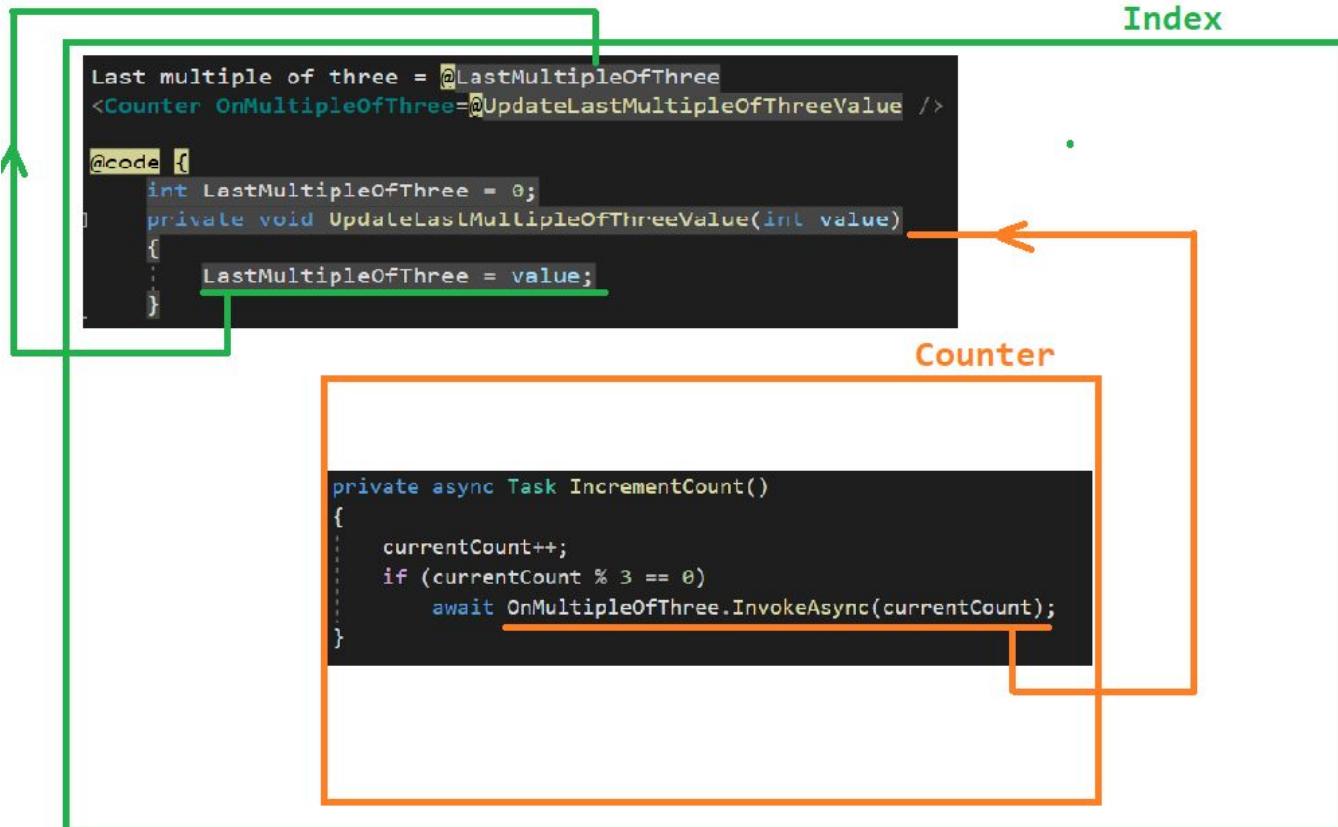
Le composant parent peut désormais prévoir une méthode qui pourra réagir à l'événement

```
Last multiple of three = @LastMultipleOfThree
<Counter OnMultipleOfThree=@UpdateLastMultipleOfThreeValue />

@code {
    int LastMultipleOfThree = 0;
    private void UpdateLastMultipleOfThreeValue(int value)
    {
        LastMultipleOfThree = value;
    }
}
```

Les événements

- Fonctionnement



Last multiple of three = 18

Counter -

Current count: 20

Click me

Les événements

A la différence des événements .NET classique , EventCallBack<T> :

- Est « Single cast » : Pas de possibilité d'ajouter +ieurs fonctions lors de l'abonnement.

```
// Setting a Blazor EventCallback  
<MyComponent SomeEvent=@MyMethodToCall/>  
  
// Setting a .NET event  
MyComponent.SomeEvent += MyMethodToCall;  
// Unscrubscribing from the event MyComponent.SomeEvent -= MyMethodToCall;
```

- Ne peut pas être null : Donc pas besoin de vérifier si il est null avant l'appel.

```
// Invoking a .NET event  
MyNetEvent?.Invoke(this, someValue);  
// Invoking a CallbackEvent<T>  
MyEventCallback.InvokeAsync(someValue);
```

Bindings

Components

Bindings

- One-Way Binding

Nous avons la possibilité de transmettre des paramètres à notre composant via l'URL.

Pour cela, nous devons définir la route pour qu'elle permette le passage de paramètre et définir une variable privée avec l'annotation [Parameter] dans le code c#. (cfr routing)

Mais il est également possible dans permettre au composant parent de transmettre un paramètre au composant enfant juste en transformant la variable privée en property.

```
DisplayCounter.razor  Imports.razor  SurveyPrompt.razor  DisplayCounter.cs  MvvmBlazor.Client
@inherits DisplayCounterBase
<h3>DisplayCounter</h3>
<p> Valeur transmise : @ParamValue</p>

public class DisplayCounterBase : ComponentBase
{
    private int _paramValue;

    [Parameter]
    public int ParamValue
    {
        get
        {
            return _paramValue;
        }

        set
        {
            _paramValue = value;
        }
    }
}
```

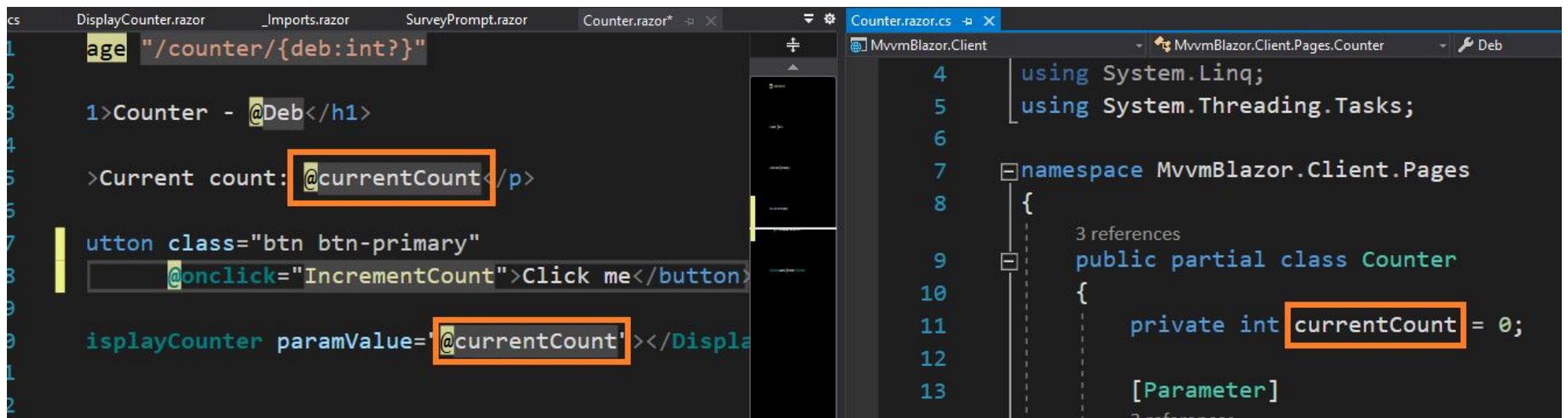
Bindings

```
Counter.razor ✘ X
@page "/counter/{deb:int?}"  
  
<h1>Counter - @Deb</h1>  
  
<p>Current count: @currentCount</p>  
  
<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>  
  
<DisplayCounter paramValue="@currentCount"></DisplayCounter>
```



Binding

En fait comme le fichier Razor, Razor.cs sont des classes, les propriétés et méthodes déclarées dans la partie c# sont accessibles via le symbole @ dans notre Html



The screenshot shows two code editors side-by-side. The left editor contains the file `Counter.razor.cs` with the following content:

```
1  page "/counter/{deb:int?}"
2
3  >Counter - @Deb</h1>
4
5  >Current count: @currentCount</p>
6
7  <button class="btn btn-primary"
8    @onclick="IncrementCount">Click me</button>
9
10 <displayCounter paramValue="@currentCount"></displayCounter>
```

The expression `@currentCount` in both the `<p>` and `<displayCounter>` tags is highlighted with a yellow rectangular selection.

The right editor contains the file `Counter.cs` with the following content:

```
4  using System.Linq;
5  using System.Threading.Tasks;
6
7  namespace MvvmBlazor.Client.Pages
8  {
9    public partial class Counter
10   {
11     private int currentCount = 0;
12
13     [Parameter]
```

The variable `currentCount` is highlighted with a yellow rectangular selection.

Two-Way Binding

Bindings

Two-Way Binding

Blazor prend également en charge la liaison de données bidirectionnelle en utilisant l'attribut bind.

```
<form>
|   Nom: <input type="text" @bind="@Mess.Nom" />
|   Email: <input type="email" @bind="@Mess.Email" />
|   Message <br />
|   <textarea @bind="Mess.MyMessage"></textarea>
|   <button @onclick="Save">Envoyer</button>
</form>
```

```
public Message Mess
{
    get
    {
        return mess;
    }

    set
    {
        mess = value;
    }
}
```

Exercice 1

Blazor

Exercice 1

Il est demandé de mettre en place une application permettant à un utilisateur de répondre à un Quizz.

Cette application devra comprendre un composant Quizz.

Ce composant Quizz :

- permet de répondre à N questions via deux boutons : Oui ou Non
- affiche le prénom transmis par son composant parent
- notifie le composant parent des réponses et de la fin du Quizz

Le design n'est pas « important » mais essayez de mettre en place les concepts de layout vue dans le support.

Exercice 1

The image displays three screenshots of a Blazor application titled "Ex1" running on localhost:44348. The application features a dark blue header bar with the title "Ex1".

- Screenshot 1:** Shows the first question: "Bonjour Mike, voici la question n° 1". Below it is the text "Blazor permet-il de faire du SPA?". At the bottom are two buttons: "Oui" and "Non".
- Screenshot 2:** Shows the second question: "Bonjour Mike, voici la question n° 2". Below it is the text "Un EventCallBack<T> fonctionne comme un .net event?". At the bottom are two buttons: "Oui" and "Non". A small "oui" response is visible below the buttons.
- Screenshot 3:** Shows the third question: "Bonjour Mike, voici la question n° 3". Below it is the text "Un EventCallBack<T> fonctionne comme un .net event?". At the bottom are two buttons: "Oui" and "Non". Two "oui" responses are visible below the buttons, followed by the text "FIN DU QUIZZ".

Forms

Blazor

Forms

Même si nous pouvons gérer les formulaires de manière classique (HTML)... Blazor nous propose un composant *EditForm*. Ce composant permet de passer un modèle qui sera binder aux inputs via un *EditContext*.

Model

Le composant *EditForm* reçoit donc un Model. Ce model peut préciser, via les DataAnnotations, les règles de validation.

```
public class UtilisateurModel
{
    private string _email, _password;

    [Required]
    [EmailAddress]
    [DataType(DataType.EmailAddress)]
    4 references
    public string Email...
    [Required]
    [DataType(DataType.Password)]
    4 references
    public string Password...
}
```

Forms

Model="Utilisateur"

Permet de lier le formulaire à la propriété du même nom dans notre .cs

Remarque:

Cette propriété doit être instancié avant la liaison

```
private UtilisateurModel _utilisateur;  
9 references  
protected UtilisateurModel Utilisateur...  
  
0 references  
public RegisterFormBase()  
{  
    _utilisateur = new UtilisateurModel();  
}
```

```
<EditForm Model="Utilisateur" >  
    <label for="Email">E-Mail</label>  
    <InputText id="Email"  
        @bind-Value="Utilisateur.Email"  
        class="form-control" />  
    <label for="Pwd">Password</label>  
    <InputText id="Pwd"  
        type="password"  
        @bind-Value="Utilisateur.Password"  
        class="form-control" />  
    <button type="submit" class="btn btn-primary">Save</button>  
</EditForm>
```

Forms

Submit

Pour détecter la soumission du formulaire, EditForm nous propose 3 events pouvant être capturé côté code Behind.

- 1) OnSubmit
Soumission du formulaire que la validation échoue ou non
- 2) OnValidSubmit
Soumission du formulaire si la validation est réussie
- 3) OnInvalidSubmit
Soumission du Formulaire si la validation échoue

Remarque :

Si OnSubmit est implémenté, les deux autres events ne seront jamais lancés

Forms

Chacun de ses événement reçoit un *EditContext* en paramètre permettant la validation supplémentaire du Model.

Afin de permettre la validation du formulaire grâce aux dataAnnotation, nous devons spécifier le composant suivant dans notre formulaire :

<DataAnnotationsValidator />

Nous pouvons également ajouter

<ValidationSummary />

Pour obtenir les informations de validation

```
<EditForm Model="Utilisateur" OnSubmit="SubmitForm">
    <DataAnnotationsValidator />
    <label for="Email">E-Mail</label>
    <InputText id="Email"
        @bind-Value="Utilisateur.Email"
        class="form-control" />
    <label for="Pwd">Password</label>
    <InputText id="Pwd"
        type="password"
        @bind-Value="Utilisateur.Password"
        class="form-control" />
    <button type="submit" class="btn btn-primary">Save</button>
    <ValidationSummary />
</EditForm>
```

Forms - OnSubmit

OnSubmit

```
<EditForm Model="Utilisateur" OnSubmit="SubmitForm">
    <DataAnnotationsValidator />
    <label for="Email">E-Mail</label>
    <InputText id="Email"
        @bind-Value="Utilisateur.Email"
        class="form-control" />
    <label for="Pwd">Password</label>
    <InputText id="Pwd"
        type="password"
        @bind-Value="Utilisateur.Password"
        class="form-control" />
    <button type="submit" class="btn btn-primary">Save</button>
    <ValidationSummary />
</EditForm>
@Status
```

Non respect des validations

Enregistrez-vous

E-Mail

Password

Save

Formulaire Submit

Enregistrez-vous

E-Mail

Password

Save

• The Email field is not a valid e-mail address.

Formulaire Submit

Forms - OnValidSubmit

OnValidSubmit

```
<EditForm Model="Utilisateur2" OnValidSubmit="SubmitValidForm">
    <DataAnnotationsValidator />
    <label for="Email">E-Mail</label>
    <InputText id="Email"
        @bind-Value="Utilisateur2.Email"
        class="form-control" />
    <label for="Pwd">Password</label>
    <InputText id="Pwd"
        type="password"
        @bind-Value="Utilisateur2.Password"
        class="form-control" />
    <button type="submit" class="btn btn-primary">Save</button>
    <ValidationSummary />
</EditForm>
<br />
@Status
```

Validation

Enregistrez-vous	Enregistrez-vous	Enregistrez-vous
E-Mail <input type="text"/>	E-Mail <input type="text" value="eed"/>	E-Mail <input type="text" value="mm@mm.be"/>
Password <input type="password"/>	Password <input type="password" value="...."/>	Password <input type="password" value="...."/>
<button type="button">Save</button>	<button type="button">Save</button>	<button type="button">Save</button>
<ul style="list-style-type: none">The Email field is required.The Password field is required.	<ul style="list-style-type: none">The Email field is not a valid e-mail address.	Formulaire ValidSubmit

Forms - OnInvalidSubmit

OnInvalidSubmit

```
<EditForm Model="Utilisateur3" OnInvalidSubmit="SubmitInValidForm">
    <DataAnnotationsValidator />
    <label for="Email">E-Mail</label>
    <InputText id="Email"
        @bind-Value="Utilisateur3.Email"
        class="form-control" />
    <label for="Pwd">Password</label>
    <InputText id="Pwd"
        type="password"
        @bind-Value="Utilisateur3.Password"
        class="form-control" />
    <button type="submit" class="btn btn-primary">Save</button>
    <ValidationSummary />
</EditForm>
@Status
```

Validation

The figure consists of three side-by-side screenshots of a Blazor application titled "Enregistrez-vous". Each screenshot shows a form with two fields: "E-Mail" and "Password". A "Save" button is at the bottom.

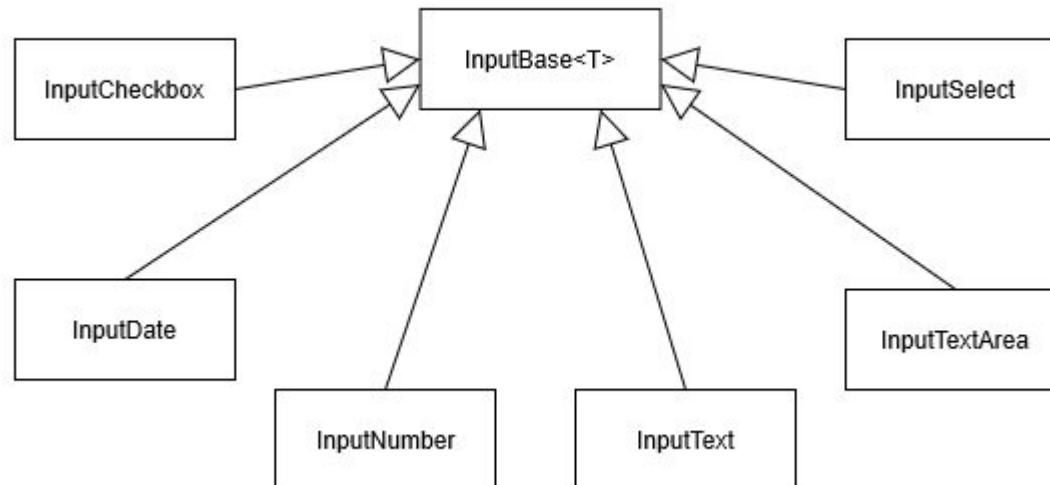
- Screenshot 1:** Both fields are empty. The "E-Mail" field has a red border and a red placeholder "dededdd". The "Password" field also has a red border and a red placeholder ".....". The "Save" button is blue.
- Screenshot 2:** The "E-Mail" field contains the invalid value "dededdd" and the "Password" field contains the invalid value ".....". Both fields have red borders and red placeholders. The "Save" button is blue.
- Screenshot 3:** The "E-Mail" field now contains a valid value "moi@lol.com" and the "Password" field contains a valid value ".....". Both fields have green borders and green placeholders. The "Save" button is blue.

Below each screenshot is a caption: "Formulaire InvalidSubmit" under the first two, and "Formulaire InvalidSubmit" under the third.

Forms

Les inputs

Il est tout à fait possible d'utiliser les inputs html classique dans le component *EditForm*. Mais il est plus aisé d'utiliser les contrôles Blazor qui ajoute des fonctionnalités telle la validation.

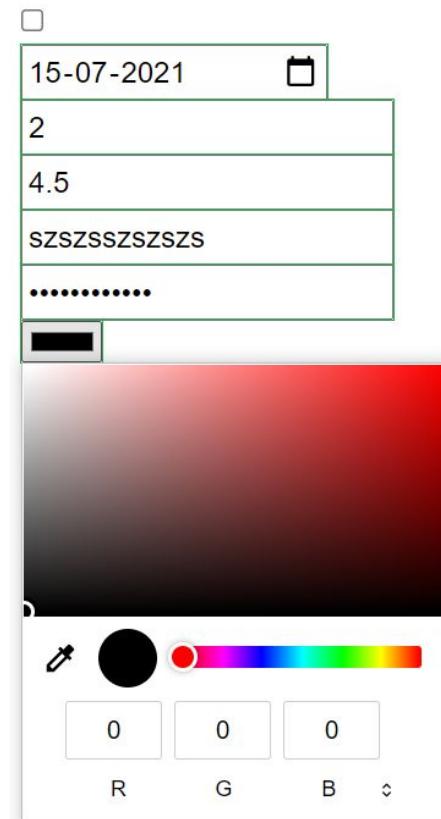


Forms

```
<EditForm Model="FormData">
    <!-- Checkbox : ne peut pas être bindé sur un élément nullable --&gt;
    &lt;InputCheckbox @bind-Value=FormData.SomeBooleanProperty /&gt;&lt;br /&gt;
    <!-- Calendrier: input type='date'. Peut être bindé sur une élément nullable
        MAIS il faut vérifier la compatibilité avec les navigateurs--&gt;
    &lt;InputDate @bind-Value=FormData.SomeDateTimeProperty
        @bind-Value:format="dd/MM/yyyy"
        ParsingErrorMessage="Doit être une date" /&gt;&lt;br /&gt;

    <!-- Peut être lié à n'importe quel format numérique. Si la valeur ne peut pas
        être parsee -&gt; Erreur validation.'--&gt;
    &lt;InputNumber @bind-Value=FormData.SomeIntegerProperty
        ParsingErrorMessage="Doit être un entier" /&gt;&lt;br /&gt;
    &lt;InputNumber @bind-Value=FormData.SomeDecimalProperty
        ParsingErrorMessage="Doit être un décimal" /&gt;&lt;br /&gt;
    <!-- Champs sans Type, nous pouvons donc ajouter type='password',
        type="color",...--&gt;
    &lt;InputText @bind-Value=FormData.SomeStringProperty /&gt;&lt;br /&gt;
    &lt;InputText type="password" @bind-Value=FormData.SomeStringProperty /&gt;&lt;br /&gt;
    &lt;InputText type="Color" @bind-Value=FormData.SomeStringProperty /&gt;&lt;br /&gt;
&lt;/EditForm&gt;</pre>
```

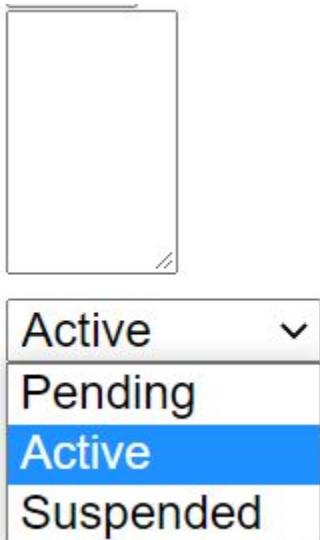
Les inputs



Forms

```
<!--TextArea -->
<InputTextArea cols="5" @bind-Value=FormData.SomeMultiLineStringProperty /><br />

<!--Select : Basé sur une énumération, la sélection se portera sur la
     valeur de la propriété Même si ce n'est pas la première'-->
<InputSelect @bind-Value=FormData.SomeSelectProperty>
    <option value="Pending">Pending</option>
    <option value="Active">Active</option>
    <option value="Suspended">Suspended</option>
</InputSelect>
```



```
public class SomeModel
{
    4 references
    public bool SomeBooleanProperty { get; set; }
    4 references
    public DateTime? SomeDateTimeProperty { get; set; }
    4 references
    public int SomeIntegerProperty { get; set; }
    4 references
    public decimal SomeDecimalProperty { get; set; }
    12 references
    public string SomeStringProperty { get; set; }
    4 references
    public string SomeMultiLineStringProperty { get; set; }
    4 references
    public SomeStateEnum SomeSelectProperty { get; set; } = SomeStateEnum.Active;
}
2 references
public enum SomeStateEnum
{
    Pending,
    Active,
    Suspended
}
```

Forms

Nous pouvons également créer nos propres inputs basés sur la classe abstraite `InputBase<T>`.

L'implémentation est simple : 1 méthode abstraite obligatoire et 1 virtual.

```
protected virtual string? FormatValueAsString(TValue? value);
```

Permet de transforme la valeur de l'input en string pour l'enregistrement futur

```
protected abstract bool TryParseValueFromString(string? value,  
[MaybeNullWhen(false)] out TValue result, [NotNullWhen(false)] out string?  
validationErrorMessage);
```

Permet de transformer la valeur string en input html

(Exemple : `InputColor` dans la solution `Formulaire_Consummation_API`)

Exercice

Exercice Formulaire :

Formulaire de contact

- Nom
- Email
- Telephone
- Message
- Choix d'une couleur

Si le formulaire est correctement rempli : on affiche à la place du formulaire un message "SUCCESS"

Si le formulaire est non correctement rempli : Afficher une alerte en rouge avec "Veuillez vérifier le formulaire"

Virtualisation

Blazor

Virtualisation

Le principe de virtualisation natif à Blazor permet l'affichage de collection générique de manière lazy-loaded.

Autrement dit, les données ne seront chargées dans le DOM qu'à la demande et en fonction des besoins.

```
<ul>
<Virtualize Items="MyList" Context="item">
|   <li>
|     @item.Name
|   </li>
</Virtualize>
</ul>
```

Le helper `<Virtualize>` applique un foreach sur l'attribut **Items**

Consommation API

Blazor

Consommation API

La consommation d'un API en Blazor suit le même principe qu'une consommation en C#.

Elle se fait via l' *HttpClient* qui est injecté dans notre Blazor et peut donc être utilisé dans nos composants.

```
2 references
public class FetchDataBase : ComponentBase
{
    [Inject]
    1 reference
    public HttpClient Http { get; set; }

    protected WeatherForecast[] forecasts;

    0 references
    protected override async Task OnInitializedAsync()
    {
        forecasts = await Http.GetFromJsonAsync<WeatherForecast[]>("WeatherForecast");
    }
}
```

Remarque : `OnInitializedAsync` est un événement qui survient après le constructeur et l'attribution des valeurs aux paramètres

Consommation API

HttpClientJsonExtensions

System.Net.Http.Json nous propose des méthodes d'extension pour faciliter la consommation d'une API.

- **GET**

- **Récupération d'une valeur typée**
- **Récupération d'une valeur dynamique**

```
response = await HttpClient.GetFromJsonAsync<UsersResponse>("https://reqres.in/api/users");

JsonElement data = await HttpClient.GetFromJsonAsync<JsonElement>("https://reqres.in/api/users");

total = data.GetProperty("total").GetInt32();
```

- **Récupération avec gestion d'erreur**

```
using var httpResponse = await HttpClient.GetAsync("https://reqres.in/invalid-url");
if (!httpResponse.IsSuccessStatusCode)
{ errorMessage = httpResponse.ReasonPhrase; return; }
UsersResponse response = await httpResponse.Content.ReadFromJsonAsync<UsersResponse>();
```

- **Récupération avec un Header**

```
HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Get, "https://reqres.in/api/users");
request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", "my-token");
request.Headers.Add("My-Custom-Header", "foobar");
using var httpResponse = await HttpClient.SendAsync(request);
UsersResponse response = await httpResponse.Content.ReadFromJsonAsync<UsersResponse>();
```

Consommation API

HttpClientJsonExtensions

System.Net.Http.Json nous propose des méthodes d'extension pour faciliter la consommation d'une API.

- **POST**

- **Envoyer un Json Body**

```
var postBody = new { Title = "Blazor POST Request Example" };
using var response = await HttpClient.PostAsJsonAsync("https://reqres.in/api/articles", postBody);
Article article = await response.Content.ReadFromJsonAsync<Article>();
```

- **Envoyer un objet typé**

```
Article art = new Article(){....};
using var response = await HttpClient.PostAsJsonAsync<Article>("https://reqres.in/api/articles", art);
Article article = await response.Content.ReadFromJsonAsync<Article>();
```

Javascript Interop

Blazor

Javascript Interop

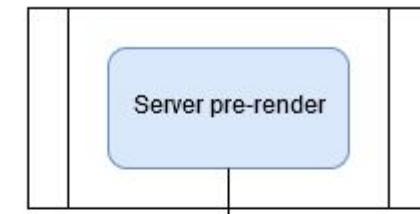
Certaines fonctionnalités ne sont pas supportées par les WebAssembly il nous faut donc passer par javascript pour y accéder.

Ordre d'initialisation

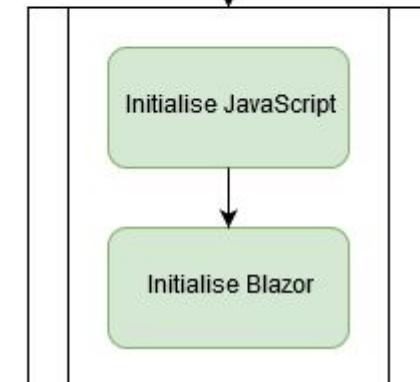
La première fois que l'utilisateur visite une URL vers notre application, Blazor affichera le composant App.razor en dehors du navigateur et enverra le code HTML résultant à celui-ci.

Après cela, JavaScript est initialisé et finalement Blazor est initialisé pour que le client puisse interagir avec celui-ci.

Server



Browser



Javascript Interop

- Appeler du JS à partir du .NET

Notre Javascript doit être ajouté soit du côté server (/Page/_Host.cshtml) soit dans du côté Client (wwwroot/index.html).

Ensuite, nous devons injecter l'interface JSRuntime dans notre composant pour pouvoir invoquer les fonctions js.



```
MyScripts.js
Info = function (message) {
    alert(message);
}

index.html
<body>
    <div id="app">Loading...</div>

    <div id="blazor-error-ui">
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">X</a>
    </div>
    <script src="_framework/blazor.webassembly.js"></script>
    <script src="js/MyScripts.js"></script>
</body>
```



```
JSComponent.razor.cs
2 references
public class JSComponentBase : ComponentBase
{
    [Inject]
    private IJSRuntime jSRuntime { get; set; }

    protected async Task SendInfo()
    {
        await jSRuntime.InvokeAsync<string>("Info", "Bonjour");
    }
}
```

Javascript Interop

Exemple d'utilisation : LocalStorage

L'accès au localstorage n'est pas possible via le webAssembly, nous devons donc passer par JS pour accéder aux éléments enregistrés (Token, User,...)

```
2 references
public class LocalStorageService: ILocalStorageService
{
    private IJSRuntime _jsRuntime;
0 references
    public LocalStorageService(IJSRuntime jsRuntime)
    {
        _jsRuntime = jsRuntime;
    }

    2 references
    public async Task<T> GetItem<T>(string key)
    {
        var json = await _jsRuntime.InvokeAsync<string>("localStorage.getItem", key);
        if (json == null)
            return default;
        return JsonSerializer.Deserialize<T>(json);
    }

    2 references
    public async Task SetItem<T>(string key, T value)
    {
        await _jsRuntime.InvokeVoidAsync("localStorage.setItem", key, JsonSerializer.Serialize(value));
    }

    1 reference
    public async Task RemoveItem(string key)
    {
        await _jsRuntime.InvokeVoidAsync("localStorage.removeItem", key);
    }
}
```

Javascript Interop

Nous pouvons ensuite utiliser notre service dans un composant

```
@page "/Storage"
@inherits StorageBase
<h3>Storage</h3>
<EditForm Model="Data">
    Clé <input type="text" @bind-value="Data.Key" />
    <br />
    Valeur <input type="text" @bind-value="Data.Value" />
    <button @onclick="AddToStorage">Ajout</button>
</EditForm>
<hr />
<EditForm Model="Data">
    Récupérer la valeur : <input type="text" value="@Data.Key" />
    <button @onclick="()=>GetData(Data.Key)">Recup</button>
</EditForm>
<div class="alert alert-primary">
    @InfoStorage
</div>
<hr />
<div class="alert alert-info">
    DEBUG : @Message
</div>
```

```
public class StorageBase : ComponentBase
{
    3 references
    protected string Message { get; set; }
    15 references
    protected Sample Data { get; set; } = new Sample();

    2 references
    protected string InfoStorage { get; set; }
    [Inject]
    2 references
    private ILocalStorageService _service { get; set; }

    void StorageBase._service.s1

    1 reference
    public async Task AddToStorage()
    {
        await _service.SetItem<string>(Data.Key, Data.Value);
        Message = $"{Data.Key} ajouté";
    }

    1 reference
    public async Task GetData(string key)
    {
        InfoStorage = await _service.GetItem<string>(Data.Key);
        Message = $"{Data.Key} récupérée";
        StateHasChanged();
    }
}
```

Exercice2

Blazor

Exercice

Mettre en place une app Blazor permettant de :

- Lister les utilisateurs
- S'enregistrer avec un email+password
- Se logger
- Atteindre une zone membre protégée nous permettant de changer notre mot de passe

Remarque :

- Utiliser une Api pour interagir avec la Base de données
- Utiliser le localstorage pour garder le statut de l'utilisateur (connecté ou non)
- Prévoir les validations nécessaires pour l'enregistrement, le login et le changement de mot de passe

Exercice

The screenshot shows a Blazor application running at `localhost:44310`. The left sidebar has a dark blue background with white text and icons. It lists "Forms - API" and "- Auth" under a header, followed by four items: "Consommation API" (selected, highlighted in grey), "Inscription", "Login", and "Zone Membre". The main content area has a light grey background. At the top, it says "Bienvenue" and "Mes utilisateurs". Below this is a table with two columns: "Email" and "Password". The table contains three rows of data:

Email	Password
User1@gmail.com	Test1234=
User2@gmail.com	User@test
User3@gmail.com	Admin

Vous pouvez partir du projet exemple présent sur https://github.com/michaelperson/Formulaire_Consummation_API.git

Sécurité

CORS (Cross-Origin Resource Sharing)

ASP.NET CORE WEB API

CORS

« La Same-Origin Policy (SOP) interdit le chargement à partir d'autres serveurs lors d'une visite d'un site Web. **Toutes les données doivent provenir de la même source, c'est-à-dire du même serveur** » (https://developer.mozilla.org/fr/docs/Web/Security/Same-origin_policy)

Nous sommes d'accord MAIS quand il s'agit d'une Web API, nous aurons majoritairement des appels venant d'une autre origine que notre serveur (Mobile Application, Angular Web Site, ...)

Nous devons donc configurer notre API pour qu'elle puisse répondre à des requêtes venant d'autres origines sous peine d'obtenir l'erreur suivante lors de la consommation

✖ Access to fetch at 'https://joke-api-strict-cors.appspot.com/r localhost/:1 random_joke' from origin '<http://localhost:3000>' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

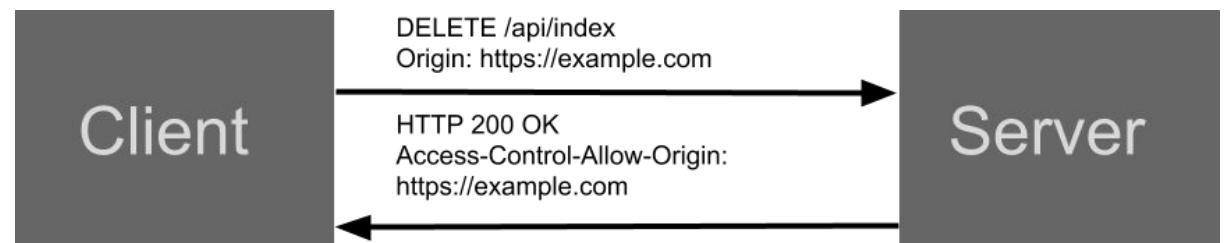
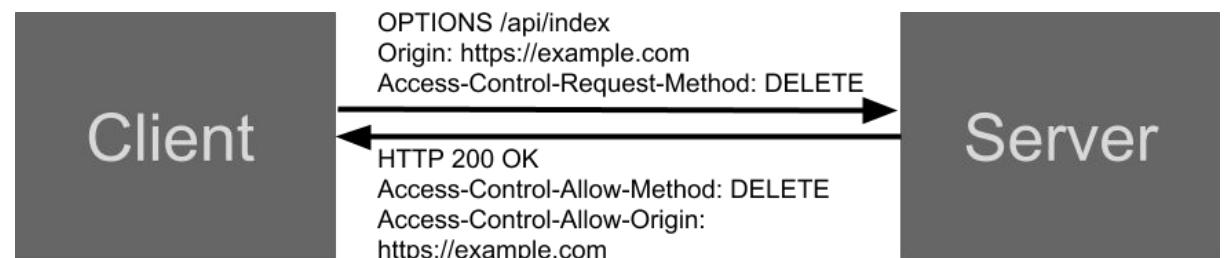
CORS

Preflight Requests

Parfois, au lieu d'une simple requête GET, un client peut avoir besoin d'envoyer des requêtes telles que PUT, DELETE, etc. Pour de telles requêtes, le navigateur envoie une requête supplémentaire (une requête OPTIONS) appelée requête Preflight.

Cela se fait juste avant la demande réelle pour s'assurer que la demande d'origine réussit.

Si c'est le cas, le navigateur envoie la demande réelle.

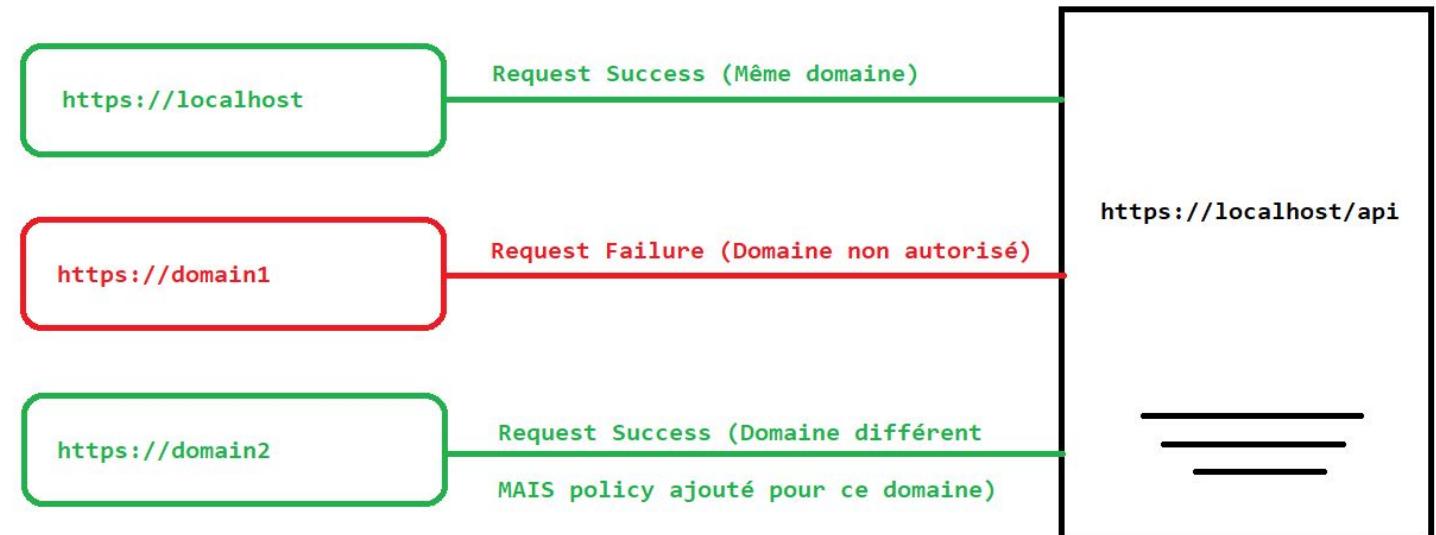


Crédit: <https://code-maze.com/>

CORS

3 façons de configurer le CORS :

1. Middleware (ConfigureService)
2. Endpoint routing (Configure)
3. [EnableCors] attribute (Controllers)



CORS

1. MiddleWare

La mise en place de CORS se fait via un policy qui va définir les règles d'accès dans la méthode *ConfigureService*.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(opt =>
    {
        opt.AddPolicy(name: _policyName, builder =>
        {
            builder.AllowAnyOrigin() ==> Toute les origines *.*  

                .AllowAnyHeader()    ==> Tous les headers (content-type, x-request-width,...)  

                .AllowAnyMethod();   ==> Tous les verbs (POST,GET,PUT, PATCH,...)
        });
    });
}
```

Et l'utiliser dans la méthode configure

```
app.UseCors(_policyName);
```

CORS

La variable *builder* est de type
Microsoft.AspNetCore.Cors.Infrastructure.Builder.

Nous pouvons donc être plus précis dans nos règles:

```
services.AddCors(opt =>
{
    opt.AddPolicy(name: _policyName, builder =>
    {
        builder.WithOrigins(new string[] { "https://localhost" })
            .WithHeaders(new string[] { "Content-Type" })
            .WithMethods(new string[] { "POST", "PUT" });
    });
});
```

```
...public class CorsPolicyBuilder
{
    ...public CorsPolicyBuilder(params string[] origins);
    ...public CorsPolicyBuilder(CorsPolicy policy);

    ...public CorsPolicyBuilder AllowAnyHeader();
    ...public CorsPolicyBuilder AllowAnyMethod();
    ...public CorsPolicyBuilder AllowAnyOrigin();
    ...public CorsPolicyBuilder AllowCredentials();
    ...public CorsPolicy Build();
    ...public CorsPolicyBuilder DisallowCredentials();
    ...public CorsPolicyBuilder SetIsOriginAllowed(Func<string, bool> isOriginAllowed);
    ...public CorsPolicyBuilder SetIsOriginAllowedToAllowWildcardSubdomains();
    ...public CorsPolicyBuilder SetPreflightMaxAge(TimeSpan preflightMaxAge);
    ...public CorsPolicyBuilder WithExposedHeaders(params string[] exposedHeaders);
    ...public CorsPolicyBuilder WithHeaders(params string[] headers);
    ...public CorsPolicyBuilder WithMethods(params string[] methods);
    ...public CorsPolicyBuilder WithOrigins(params string[] origins);
}
```

CORS

Nous pouvons également utiliser l'instruction `AddDefaultPolicy` pour configurer les règles par défaut sans nommer la policy.

```
services.AddCors(opt =>
{
    opt.AddDefaultPolicy( builder =>
    {
        builder.WithOrigins(new string[] { "https://localhost" })
            .WithHeaders(new string[] { "Content-Type" })
            .WithMethods(new string[] { "POST", "PUT" });
    });
});
```

CORS

2. Endpoint Routing

La mise en place de CORS peut se faire également par route.

!!! Ne supporte pas les pre-flight request!!!

Pour ce faire, nous utiliserons la méthode d'extension `RequireCors` en complément de `AddPolicy` ou en spécifiant la règle directement comme paramètre.

```
        app.UseCors(_policyName);

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers().RequireCors(_policyName);
    endpoints.MapBlazorHub().RequireCors(builder =>
{
        builder.WithOrigins(new string[] { "https://mondomaine" })
            .AllowAnyHeader()
            .WithMethods(new string[] { "POST", "PUT" });
    });
});
```

CORS

Nous pouvons cependant configurer la *Preflight request* via l'attribut `[HttpOptions]` que nous plaçons au dessus d'un méthode afin de configurer celle-ci.

```
// OPTIONS: api/WeatherForecast/5
[HttpOptions("{id}")]
0 references
public IActionResult PreflightRoute(int id)
{
    return NoContent();
}

// OPTIONS: api/WeatherForecast
[HttpOptions]
0 references
public IActionResult PreflightRoute()
{
    return NoContent();
}

[HttpPut("{id}")]
0 references
public IActionResult PutTodoItem(int id)
{
    if (id < 1)
    {
        return BadRequest();
    }
    ...
}
```

CORS

3. [EnableCors] attribute (Controllers)

Si nous ne désirons activer le CORS que pour certaines routes, il est également possible d'utiliser l'attribut `[EnableCors]` au niveau du controller complet ou des actions :

- Soit juste l'attribut `[EnableCors]` qui appliquera la policy par défaut
- Soit `[EnableCors(" Policy Name ")]` qui permet d'appliquer des politiques de cors différents suivant les routes

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = false, Inherited = true)]
public class EnableCorsAttribute : Attribute, IEnableCorsAttribute, ICorsMetadata
{
    ...
    public EnableCorsAttribute();
    public EnableCorsAttribute(string policyName);

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

CORS

Startup.cs

```
private readonly string _policyName = "MyCorsPolicy";
0 references
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(opt =>
    {
        opt.AddPolicy(name: _policyName, builder =>
        {
            builder.WithOrigins(new string[] { "https://localhost" })
                .WithHeaders(new string[] { "Content-Type" })
                .WithMethods(new string[] { "POST", "PUT" });
        });
    });
}
```

Un controller

```
[HttpPost("{id}")]
[EnableCors("MyCorsPolicy")]
0 references
public IActionResult PutTodoItem(int id)
{
    if (id < 1)
    {
        return BadRequest();
    }
    ...
}
[HttpPost("{id}")]
[EnableCors("MyCorsPolicy")]
0 references
public IActionResult PostTodoItem(int id, string Name, float value)
{
    if (id < 1)
    {
        return BadRequest();
    }
    ...
}
[ApiController]
[EnableCors]
[Route("[controller]")]
3 references
public class WeatherForecastController : ControllerBase
```

AuthenticationStateProvider

To do

Demo : https://github.com/SteveBstorm/DemoAuthenticationStateProvider_Client

AuthenticationStateProvider

La sécurité en Blazor Client s'effectue au travers d'une classe héritant de **AuthenticationStateProvider**, et permet de propager l'état de connexion utilisateur sous forme de *ClaimPrincipal*.

```
public class MyAuthStateProvider : AuthenticationStateProvider
```

L'état ainsi propagé sera interprété par *[AuthorizeView]* pour limiter les accès utilisateurs à certaines portions de l'application.

AuthenticationStateProvider

La méthode *GetAuthenticationStateAsync()* retourne une *Task<AuthenticationState>* propagée à l'ensemble de l'application.

```
Task<AuthenticationState> GetAuthenticationStateAsync()
```

Il est nécessaire d'implémenter cette méthode pour que le système identifie qu'un utilisateur est bien authentifié.

AuthorizeView

L'attribut *AuthorizeView* sera positionné sur chaque composant Razor nécessitant une limitation d'accès. Et pourra prendre deux états :

```
<Authorized>
| ...
</Authorized>
```

```
<NotAuthorized>
| ...
</NotAuthorized>
```

Le contenu de ces balises sera utilisé en fonctions du State fourni par notre *AuthStateProvider*.

AuthorizeView

Les informations sur l'utilisateur courant retourné par la méthode *GetAuthenticationStateAsync()* seront accessible dans l'application via :

`@context.User` Qui représente l'objet `HttpContext` au sein des composants

Il suffira dès lors de parcourir les claims contenus dans l'objet User.

Design Pattern

Injection de dépendances

LE PRINCIPE D'INJECTION DE DÉPENDANCES

```
public class Startup
{
    1 reference
    public IConfiguration Configuration { get; }
    0 references
    public Startup(IConfiguration configuration)...
    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();
    }
    0 references
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)...
}
```

Depuis le Framework Core 3.1, Microsoft a ajouté sa propre bibliothèque de classes pour l'injection de dépendances et l'a publiée sur « Nuget » sous le nom : « Microsoft.Extensions.DependencyInjection ».

Dans le cadre des applications web .Net Core (Asp et Api) celui-ci est déjà intégré dans le fichiers « Startup.cs » au travers de la méthode « ConfigureServices ».



Exemple complet :
[TMCognitic/DemoDependencyInjection \(github.com\)](https://github.com/TMCognitic/DemoDependencyInjection)

LE PRINCIPE D'INJECTION DE DÉPENDANCES

Le container de type « `IServiceCollection` » nous propose trois méthodes de base :

1. `AddSingleton` :
Retourne toujours la même instance d'objet
2. `AddScoped` :
Dans le même scope, retourne toujours la même instance d'objet
3. `AddTransient` :
Retourne toujours une nouvelle instance d'objet

Quant au type « `IServiceProvider` » il propose la méthode « `GetService` ».

```
public class Startup
{
    public IConfiguration Configuration { get; }

    public Startup(IConfiguration configuration) {}

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();
        services.AddSingleton<ISingletonRepository, SingletonService>();
        services.AddScoped<IScopedRepository, ScopedService>();
        services.AddTransient<ITransientRepository, TransientService>();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {}
}
```

LE PRINCIPE D'INJECTION DE DÉPENDANCES

```
public class HomeController : Controller
{
    private readonly ISingletonRepository _singletonRepository;
    private readonly IScopedRepository _scopedRepository;
    private readonly ITransientRepository _transientRepository;

    public HomeController(ISingletonRepository singletonRepository,
                          IScopedRepository scopedRepository,
                          ITransientRepository transientRepository)
    {
        _singletonRepository = singletonRepository;
        _scopedRepository = scopedRepository;
        _transientRepository = transientRepository;
    }

    public IActionResult Index()
    {
        ViewBag.SingletonHashCode = _singletonRepository.GetHashCode();
        ViewBag.ScopedHashCode = _scopedRepository.GetHashCode();
        ViewBag.TransientHashCode = _transientRepository.GetHashCode();
        return View();
    }

    public IActionResult Privacy()...
    Masqué pour des raisons de mise en page
    public IActionResult Error()...
}
```

Pour injecter une ressource à un contrôleur, nous passerons par son constructeur.

LE PRINCIPE D'INJECTION DE DÉPENDANCES

Pour injecter une ressource à un vue, nous utiliserons « @inject ».

Pour rappel, si notre ressource doit être utilisée dans toute nos pages, nous pouvons le déclarer dans le fichier « _ViewImports.cshtml ».

Il reste à noter qu'injecter un service d'accès au données directement dans une vue va à l'encontre de tout ce que nous avons vu en terme de séparation de couche du « MVC ».

```
@inject ISingletonRepository singletonRepository  
@inject IScopedRepository scopedRepository  
@inject ITransientRepository transientRepository  
  
{@  
    ViewData["Title"] = "Home Page";  
}  
  
<table class="table">  
  <thead>  
    <tr>  
      <th>...</th>  
      <th>...</th>  
      <th>...</th>  
    </tr>  
  </thead>  
  <tbody>  
    <tr>  
      <td>Singleton</td>  
      <td>@ViewBag.SingletonHashCode</td>  
      <td>@singletonRepository.GetHashCode()</td>  
    </tr>  
    <tr>...</tr>  
    <tr>...</tr>  
  </tbody>  
</table>
```