

# **C#, DÉVELOPPER EN .NET AVEC VISUAL STUDIO**

INTRODUCTION À MICROSOFT .NET



# SOMMAIRE

## **MODULE#01 .NET Core**

Leçon 1 Introduction

Leçon 2 Démarrage

Leçon 3 ConfigureServices

## **MODULE#02 Injection de dépendances**

Leçon 1 Introduction

Leçon 2 Durées de vie du service

## **MODULE#03 Introduction à Blazor**

## **MODULE#04 Les composants Blazor**

## **MODULE#05 Le Render Tree**

# SOMMAIRE

MODULE#06 Les templates de composants

MODULE#07 Le Routing

MODULE#08 Les formulaires

MODULE#09 L'intérop JS

MODULE#10 La sécurité Blazor

MODULE#11 Le déploiement Blazor

MODULE#12 Les contrôleur API

MODULE#13 EF Core

# PRÉSENTATIONS ET TOUR DE TABLE



# MODULE#01

.NET CORE



# LEÇON#01

INTRODUCTION

# INTRODUCTION

*Framework multiplateforme à hautes performances et open source*

- Applications et Services web, applications IoT et backends mobiles.
- Windows, MacOS et Linux.
- Environnement cloud ou local
- Exécution sur .NET Core et .Net Standard

## AVANTAGES

- Un scénario unifié pour créer une interface utilisateur web et des API web.
- Conçu pour la testabilité.
- Razor Pages permet de coder des scénarios orientés page de façon plus simple et plus productive.
- Capacité à développer et à exécuter sur Windows, MacOS et Linux.
- Open source et centré sur la communauté.
- L'intégration de Framework modernes, côté client et des workflows de développement



## AVANTAGES

- Un environnement prêt pour le cloud et basé sur des fichiers de configuration système.
- Une Injection de dépendances intégrée.
- Un pipeline de requête HTTP léger, haute performance et modulaire.
- La capacité à héberger sur IIS, Nginx, Apache, Docker, ou d'un auto-hébergement dans votre propre processus.
- La gestion de version des applications côte à côte lorsque la cible est .NET Core.
- SignalR (WebSocket) / WASM (Blazor)

# NOUVEAUTÉS DE ASP.NET CORE

## *SignalR*

- SignalR a été réécrit pour ASP.NET Core 2.1. ASP.NET Core
- SignalR inclut un certain nombre d'améliorations :
  - Un modèle simplifié de montée en puissance parallèle.
  - Un nouveau client JavaScript sans dépendance de jQuery.
  - Un nouveau protocole binaire compact basé sur MessagePack.
  - Prise en charge des protocoles personnalisés.
  - Un nouveau modèle réponse de streaming.
  - Prise en charge des clients basés sur des WebSocket nus.

# NOUVEAUTÉS DE ASP.NET CORE

## *Bibliothèques de classes Razor*

- Le temps de démarrage de l'application est nettement plus rapide.
- Les mises à jour rapides des pages et vues Razor au moment de l'exécution sont toujours disponibles dans le cadre d'un flux de travail de développement itératif.

## *HTTPS*

- Https est actif par défaut

## *RGPD*

- ASP.NET Core fournit des API et des modèles qui aident à satisfaire à certaines des exigences du RGPD

# NOUVEAUTÉS DE ASP.NET CORE

## *Tests d'intégration*

- Un nouveau package est introduit qui simplifie la création et l'exécution de tests. Le package *Microsoft.AspNetCore.Mvc.Testing*

## *[ApiController], ActionResult<T>*

- NET Core ajoute de nouvelles conventions de programmation qui facilitent la génération d'API web propres et descriptives

# NOUVEAUTÉS DE ASP.NET CORE

## *IHttpClientFactory*

- ASP.NET Core 2.1 inclut un nouveau service *IHttpClientFactory* qui facilite la configuration et l'utilisation d'instances de *HttpClient* dans les applications

## *Configuration du transport Kestrel*

- Dans ASP.NET Core 2.1, le transport par défaut de Kestrel n'est plus basé sur Libuv, mais sur des sockets managés

# NOUVEAUTÉS DE ASP.NET CORE

## *Générateur d'hôte générique*

- Le générateur d'hôte générique (HostBuilder) a été introduit. Ce générateur peut être utilisé pour les applications qui ne traitent pas les requêtes HTTP (messagerie, les tâches en arrière-plan, etc.).

## *Modèles SPA mis à jour*

- Les modèles d'applications monopages pour Angular, React et React avec Redux sont mis à jour pour utiliser les systèmes de génération et les structures de projet standard pour chaque Framework.

QUESTIONS? REMARQUES?



# LEÇON#02

DÉMARRAGE



# CLASSE STARTUP

Utilisation de la classe *Startup* pour la configuration et l'exécution de l'application

- *ConfigureServices* permettant de configurer les services de l'application.
- *Configure* pour créer le pipeline de traitement de demande de l'application.

```
public class Startup
{
    0reference
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    1reference
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    0reference
    public void ConfigureServices(IServiceCollection services) {...}

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    0reference
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {...}
}
```

## ENREGISTREMENT DE LA CLASSE STARTUP

La classe Startup doit être enregistré dès le démarrage de l'application

```
public class Program
{
    0 references
    public static void Main(string[] args) ...

    1 reference
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

# UTILISATION COURANTE DE L'INJECTION DE DÉPENDANCE

Une utilisation courante de l'injection de dépendances :

- *IWebHostEnvironment* pour les informations liées à l'environnement d'exécution
- *IConfiguration* pour lire la configuration de l'application
- *ILoggerFactory* pour créer un enregistreur d'événements

```
public class UserService
{
    private readonly ILoggerFactory loggerFactory;
    private readonly IConfiguration configuration;
    0 references
    public UserService(ILoggerFactory factory, IConfiguration config)
    {
        loggerFactory = factory;
        configuration = config;
    }

    0 references
    public User GetUser()
    {
        var logger = loggerFactory.CreateLogger<UserService>();

        logger.LogInformation("Récupération de l'âge dans la configuration");
        var age = configuration.GetValue<int>("Age");
        logger.LogInformation("Création d'un utilisateur");
        return new User() { Age = age, Firstname = "Jean", Lastname = "Martin" };
    }
}
```

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }
}
```

QUESTIONS ? REMARQUES ?



# LEÇON#03

CONFIGURESERVICES

# MÉTHODE CONFIGURESERVICES

La méthode **ConfigureServices** est :

- Appelée par l'hôte web avant la méthode Configure pour configurer les services de l'application.
- L'emplacement où les options de configuration sont définies.

```
public void ConfigureServices(IServiceCollection services)
{
    //Add framework services
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDatabaseDeveloperPageExceptionFilter();

    services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddControllersWithViews();

    //Add application services
    services.AddTransient<EmailService>();
    services.AddTransient<UserService>();
}
```

Par default, on appelle toutes les méthodes d'ajout de services **Add**{service} puis leurs configurations **Configure**{service}

# MÉTHODE CONFIGURE

La méthode *Configure* est utilisée pour spécifier la façon dont l'application répond aux requêtes HTTP

Composants intergiciels  
(Middleware)

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseMigrationsEndPoint();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
        endpoints.MapRazorPages();
    });
}
```

## MÉTHODES PRATIQUES

Des méthodes pratiques comme *ConfigureServices()* ou *ConfigureLogging()* peuvent être utilisées au lieu de spécifier une classe *Startup*

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        })
        .ConfigureServices(services => {
            services.AddRazorPages();
            services.AddAuthentication();

            services.AddScoped<UserService>();
        });
```



QUESTIONS ? REMARQUES ?



# **MODULE#02**

INJECTION DE DÉPENDANCES



# LEÇON#01

## INTRODUCTION

# INTRODUCTION

ASP.NET Core prend en charge le modèle de conception de logiciel avec injection de dépendances (DI).

Une technique permettant d'obtenir une inversion de contrôle (IoC) entre les classes et leurs dépendances.

- Une *dépendance* est un objet qui nécessite un autre objet

```
public class Calculator
{
    public Produit Produit { get; set; }

    References
    public decimal Calculate()
    {
        return Produit.Prix * Produit.Qte;
    }
}
```

La dépendance est injectée par le constructeur

```
public class Calculator
{
    public IProduit Produit { get; set; }

    References
    public decimal Calculate()
    {
        return Produit.Prix * Produit.Qte;
    }

    References
    public Calculator(IProduit prod)
    {
        Produit = prod;
    }
}
```

# IOC (INVERSION OF CONTROL)

ASP.NET Core fournit un conteneur de service intégré, `IServiceProvider`. Les services sont inscrits dans la méthode `Startup.ConfigureServices` de l'application.

```
public class Startup
{
    0 references
    public Startup(IConfiguration configuration) ...

    2 references
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddTransient<IProduit, Produit>();
    }
}
```

La dépendance, peut avoir elle même des dépendances.

ASP.NET core va se charger d'injecter les dépendances dans le constructeur

```
public class Produit : IProduit
{
    private readonly ILogger<Produit> logger;

    0 references
    public Produit(ILogger<Produit> log)
    {
        logger = log;
    }

    0 references
    public string Libelle { get; set; }
    2 references
    public int Prix { get; set; }
    2 references
    public int Qte { get; set; }
}
```

# SERVICES FOURNIS PAR LE FRAMEWORK

## Type de service

Microsoft.AspNetCore.Hosting.Builder.IApplicationBuilderFactory  
Microsoft.AspNetCore.Hosting.IApplicationLifetime  
Microsoft.AspNetCore.Hosting.IHostingEnvironment  
Microsoft.AspNetCore.Hosting.IStartup  
Microsoft.AspNetCore.Hosting.IStartupFilter  
Microsoft.AspNetCore.Hosting.Server.IServer  
Microsoft.AspNetCore.Http.IHttpContextFactory  
Microsoft.Extensions.Logging.ILogger<T>  
Microsoft.Extensions.Logging.ILoggerFactory  
Microsoft.Extensions.ObjectPool.ObjectPoolProvider  
Microsoft.Extensions.Options.IConfigureOptions<T>  
Microsoft.Extensions.Options.IOptions<T>  
System.Diagnostics.DiagnosticSource  
System.Diagnostics.DiagnosticListener

## Durée de vie

*Transient*  
*Singleton*  
*Singleton*  
*Singleton*  
*Transient*  
*Singleton*  
*Transient*  
*Singleton*  
*Singleton*  
*Transient*  
*Singleton*  
*Singleton*  
*Singleton*

QUESTIONS? REMARQUES?



# LEÇON#02

DURÉES DE VIE DU SERVICE



## LES DIFFÉRENTS TYPES DE SERVICE

### *Transient*

- Des services à durée de vie temporaire (Transient) sont créés chaque fois qu'ils sont demandés. Cette durée de vie convient parfaitement aux services légers et sans état.

### *Scoped*

- Les services à durée de vie délimitée (Scoped) sont créés une seule fois par requête.

### *Singleton*

- Les services avec une durée de vie singleton sont créés la première fois qu'ils sont demandés. Chaque requête ultérieure utilise la même instance.

# INSCRIPTIONS

L'inscription des services se fait dans la classe  
*ConfigureServices*

```
public class Startup
{
    0 references
    public Startup(IConfiguration configuration)...

    2 references
    public IConfiguration Configuration { get; }

    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddTransient<IProduit, Produit>();
        services.AddScoped<IProduit, Produit>();
        services.AddSingleton<IProduit, Produit>();
    }
}
```

ou

ou

## APPELER DES SERVICES À PARTIR DE MAIN

Créez un *IServiceScope* avec *IServiceScopeFactory.CreateScope* pour résoudre un service Scoped dans le scope de l'application. Cette approche est pratique pour accéder à un service Scoped au démarrage pour exécuter des tâches d'initialisation

```
public static void Main(string[] args)
{
    var host = CreateHostBuilder(args).Build();

    using (var serviceScope = host.Services.CreateScope())
    {
        var services = serviceScope.ServiceProvider;

        try
        {
            var serviceContext = services.GetRequiredService<IProduit>();
            //Use the context here
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred");
            throw;
        }
    }

    host.Run();
}
```

## CONCEPTION DE SERVICES

Les bonnes pratiques :

- Concevoir des services afin d'utiliser l'injection de dépendances pour obtenir leurs dépendances.
- Éviter les appels de méthode statiques avec état (une pratique appelée static cling).
- Éviter une instantiation directe de classes dépendantes au sein de services. L'instantiation directe associe le code à une implémentation particulière.

# CONCEPTION DE SERVICES

Les bonnes pratiques :

- Respecter les principes SOLID
  - S – Single Responsibility Principle
  - O – Open/Closed Principle
  - L – Liskov Substitution Principle
  - I – Interface Segregation Principle
  - D – Dependency Inversion Principle

# SUPPRESSION DES SERVICES

Le conteneur d'injection de dépendance ne supprime pas les instances ajoutées manuellement par le code utilisateur (développeur).

```
public class Service1 ...  
1 reference  
public class Service2 ...  
1 reference  
public class Service3 ...  
2 references  
public class Startup  
{  
    0 references  
    public Startup(IConfiguration configuration) ...  
    1 reference  
    public IConfiguration Configuration { get; }  
  
    0 references  
    public void ConfigureServices(IServiceCollection services)  
    {  
        services.AddRazorPages();  
        services.AddAuthentication();  
  
        //The container creates the following instances and disposes them automatically.  
        services.AddScoped<Service1>();  
        services.AddSingleton<Service2>();  
  
        //The container doesn't create the following instances, so it doesn't dispose the instance  
        services.AddSingleton(new Service3());  
    }  
}
```

QUESTIONS ? REMARQUES ?



# DÉMONSTRATION





# MODULE#03


INTRODUCTION À BLAZOR



## QU'EST-CE QUE BLAZOR ?

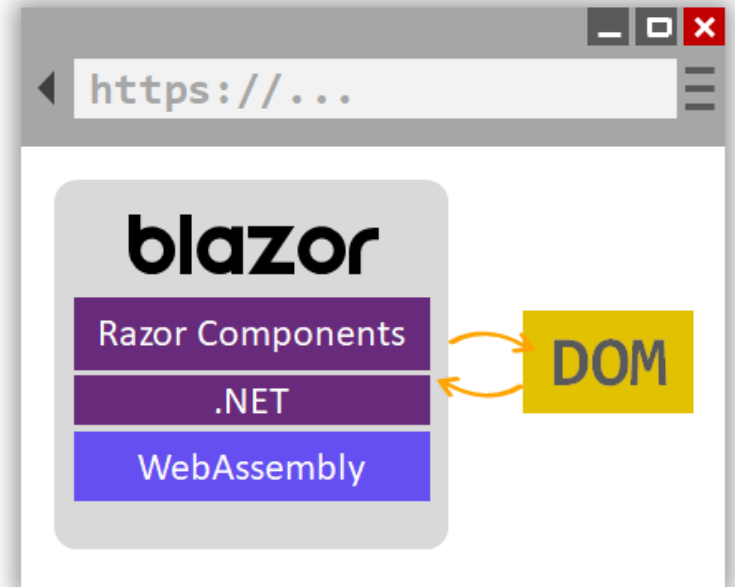
- Framework de développement pour Single Page Application (SPA)
- Blazor = Combinaison de Browser et Razor
- Ne requière aucun plugin navigateur pour fonctionner
- Blazor est Open-source
- Utilise les librairie .NET pour le développement côté client.
- Permet d'écrire du code C# à la place de Javascript
- S'intègre à des plateformes d'hébergement modernes (Docker, ...)
- Il existe deux mode d'hébergement de Blazor
  - **Blazor WebAssembly**
  - **Blazor Server**

## WEBASSEMBLY, QUESACO ?

- WebAssembly (wasm) :  
« C'est un langage bas niveau type assembleur avec un format binaire compact fournissant des performances quasi-natives permettant d'exécuter du code tel que C/C++, C# et Rust dans les navigateurs modernes. »
  - C'est un standard du W3C
  - WASM fonctionne en parallèle avec Javascript
- 

## BLAZOR WEBASSEMBLY

- Blazor WebAssembly est la technologie Microsoft exploitant WASM
- Permet de développer d'exécuter du code C# à l'aide de Razor page dans les navigateurs adhérents au standard WASM.
- Code compilé → navigateur
- Optimise la taille de l'app
- Runtime utilise javascript pour gérer les manipulations du DOM

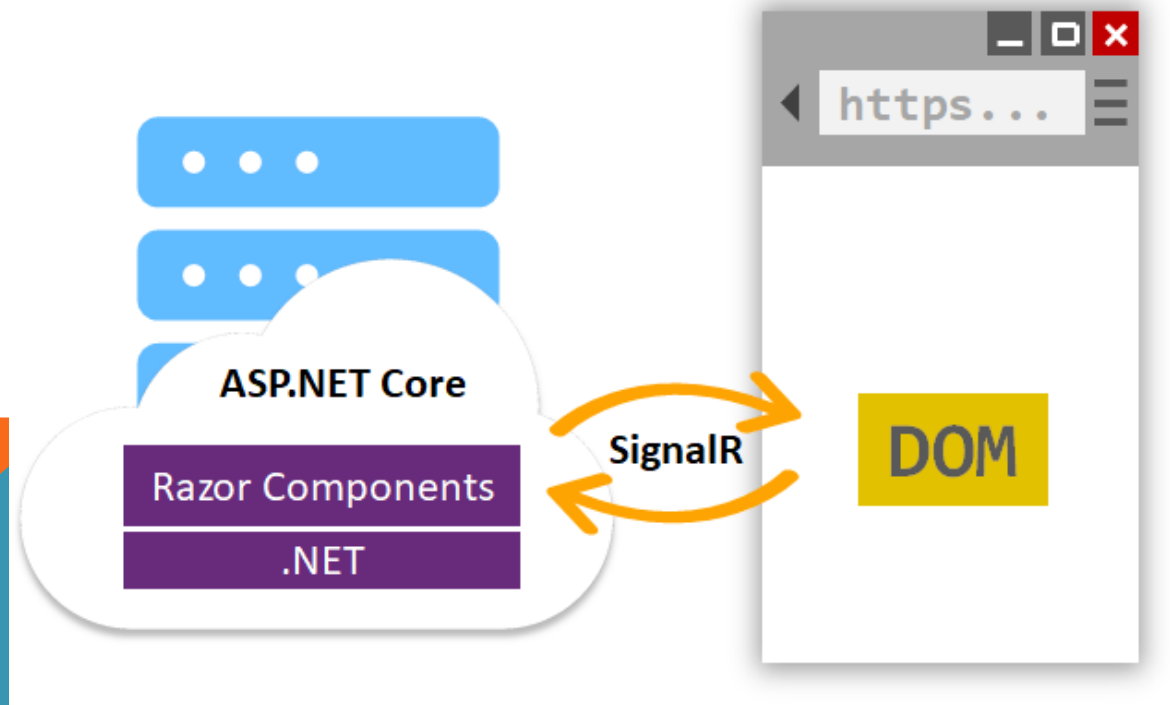


## PROS/CONS DE BLAZOR WASM

- PROS:
  - Fonctionne à l'intérieur du navigateur (création d'une PWA)
  - Peut fonctionner hors ligne
  - Exécuté sur la machine du client => Charge réduite du serveur
- CONS :
  - Nécessite de télécharger les DLL .NET
  - Temps de démarrage plus long
  - Moins performant que Blazor Server (Compilation Ahead-of-time prévu amélioré les performances de Mono)
  - Actuellement Mono-threader
  - Ne fonctionne que sur les navigateurs modernes

## BLAZOR SERVER

- Blazor Server découple la logique de rendu des composants et la façon dont les mises à jour de l'IHM sont appliquées
- Permet l'hébergement de composants razor dans une app .NET Core
- Gère les mises à jour de l'IHM à l'aide d'une connexion SignalR



## PROS/CONS DE BLAZOR SERVER

- PROS:
  - Pré-rendu du HTML avant de l'envoyer vers le navigateur
  - Compatible avec les moteurs de recherche
  - Aucun temps de démarrage perceptible
  - Fonctionne sur des navigateurs plus anciens
- CONS :
  - Mise en mémoire de la session utilisateur et utilise SignalR pour communiqué avec le server
    - Coût en ressource par utilisateur non négligeable
    - Lié au serveur qui a répondu en premier (pas de load-balancing)
  - Nombreux échange client-server = une bonne connexion

# INSTALLATION DE BLAZOR

- Disponible à partir de .NET Core 3.2 et VS(2019) 16.6
- Beaucoup de nouveauté avec .NET 5
  - VS(2019) 16.8 minimum pour .NET 5
- Contenu dans les outils de développement web ASP.NET





# CREATION D'UN PROJET BLAZOR



## Blazor App

Project templates for creating Blazor apps that run on the server in an ASP.NET Core app or in the browser on WebAssembly (wasm). These templates can be used to build web apps with rich dynamic user interfaces (UIs).

[C#](#)[Linux](#)[macOS](#)[Windows](#)[Cloud](#)[Web](#)

## Create a new Blazor app

.NET 5.0



### Blazor Server App

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



### Blazor WebAssembly App

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

### Authentication

No Authentication

[Change](#)

### Advanced

☒ Configure for HTTPS

☐ Enable Docker Support

(Requires [Docker Desktop](#))

Linux

Author: Microsoft

Source: Templates 5.0.0

[Get additional project templates](#)

Back

Create

# CRÉER UNE PAGE BLAZOR

```
@page "/counter"
```

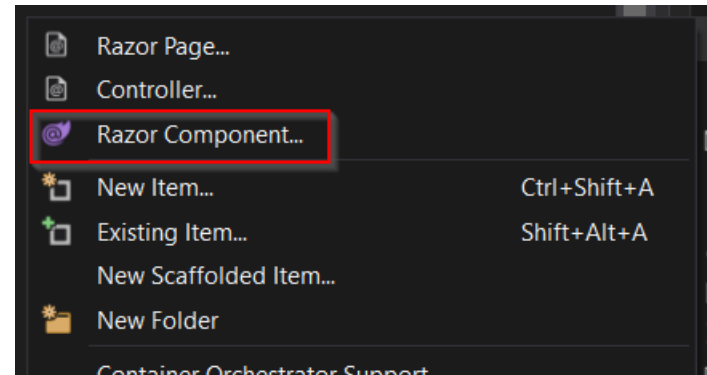
```
<h1>Counter</h1>
```

```
<p>The counter value is: @counter</p>
```

```
@code {
```

```
    private int counter = 42;
```

```
}
```



QUESTIONS ? REMARQUES ?



# DÉMONSTRATION



# MODULE#04

LES COMPOSANTS BLAZOR



# **LES LAYOUTS**



# CRÉATION D'UN LAYOUT BLAZOR

- Tout modèle de layout nécessite de d'hériter de la classe LayoutComponentBase

**@inherits** LayoutComponentBase

```
<div class="main">
  <header>
    <h1>This is the header</h1>
  </header>
  <div class="content px-4">
    @Body
  </div>
  <footer>
    This is the footer
  </footer>
</div>
```

# CRÉATION D'UN LAYOUT BLAZOR

- L'application Blazor correspond au contenu de la balise <app>

index.html

Layout

Page

```
<html>
  ▶ <head>...</head>
  ▼ <body>
    ▼ <app>
      ▼ <div class="main">
        ▼ <header>
          <h1>This is the header</h1>
        </header>
        <div class="content">
          This is the content of your embedded page!
        </div>
        <footer>
          This is the footer
        </footer>
      </div>
    </app>
    <script src="_framework/blazor.webassembly.js"></script>
    <script src="_framework/wasm/mono.js" defer></script>
  </body>
</html>
```



# CRÉATION DE LAYOUT BLAZOR

- L'utilisation d'un layout se fait à l'aide de :

**@layout** MainLayout

- Le layout par défaut des pages est défini dans app.razor

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

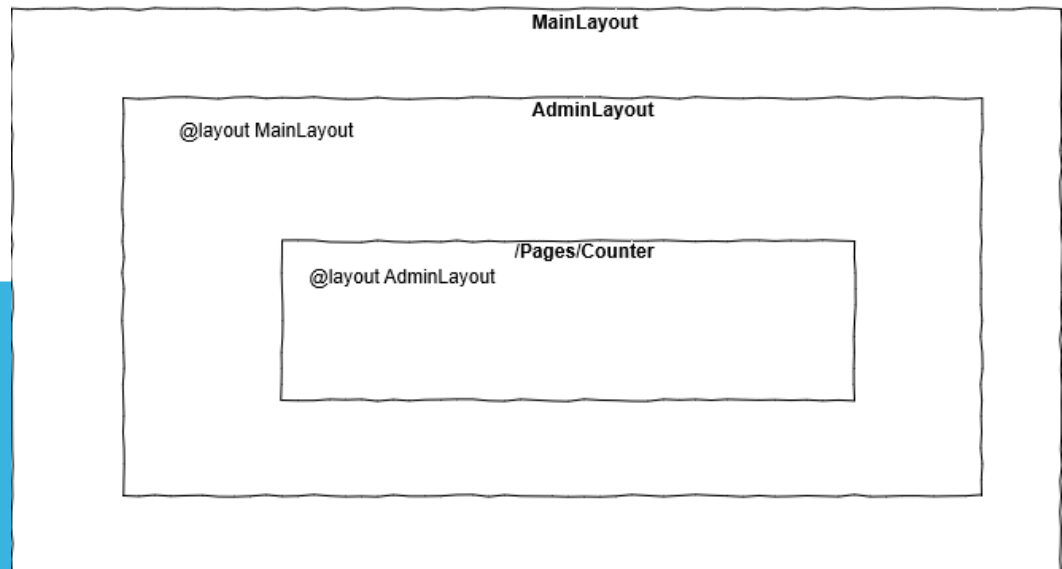
- Création de layouts imbriqués en définissant un nouveau layout qui fait appel à @layout

**@inherits** LayoutComponentBase

**@layout** MainLayout

```
<h1>Admin</h1>
```

**@Body**



## **LES COMPOSANTS BLAZOR**



# CRÉATION DE COMPOSANT BLAZOR

- Ajout d'un nouveau composant

`<h3>MyComponent</h3>`

```
@code {  
}
```

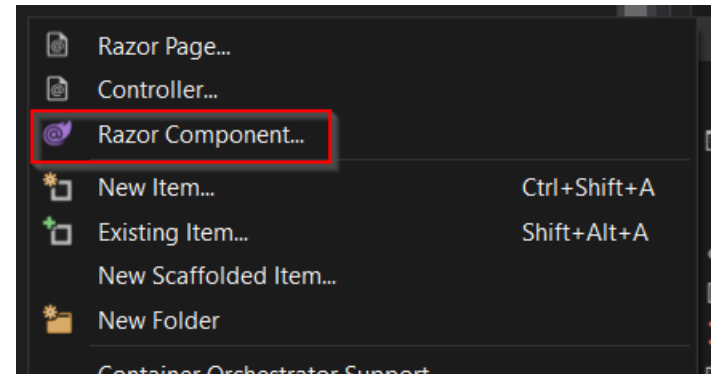
- Les composants se référencent dans une razor page ou dans un autre composant comme ceci

`<MyComponent></MyComponent>`

- Si le composant est dans un autre namespace, on peut utiliser un using dans la page

```
@using BlazorApp2.Components
```

- Il est aussi possible d'ajouter le using dans \_Imports.razor pour que tous les composants connaissent le using



## ONE-WAY BINDING

- Il est possible d'afficher directement la valeur d'une propriété c# dans le code HTML

```
<div>
```

```
    CurrentValue in MyComponent is @CurrentValue
```

```
</div>
```

```
@code {
```

```
    public int CurrentValue { get; set; }
```

```
}
```

- Il est possible de transmettre un paramètre à vos composants en décorant une propriété de l'attribut [Parameter] et en le transmettant au composant

```
<MyComponent CurrentValue=@currentCount></MyComponent>
```

# LITTÉRALE, EXPRESSION ET DIRECTIVE

- Valeur en paramètre d'un composant :  
Type simple(int, string, etc), objet, func<T>, actions, etc...
- Blazor interprète la valeur transmise en fonction de la présence ou non du symbol @
  - Param = 42 → Littérale
  - Param = @Value → Expression
  - @Param = Value → Directive
- Les balises Blazor autorise le choix de mettre ou non des parenthèses

```
<MyComponent Value=@SomeValue />
```

```
<MyComponent Value="@SomeValue" />
```

# LITTÉRALE ET EXPRESSION

- Une valeur passée sans le @ est considérée comme valeur littérale

Type	Razor	HTML
Attribut HTML	<input size=8/>	<input size="8"/>
Component parameter	<MyHeader Text="Hello" Visible=true/>	<h1>Hello</h1>

- Pour faire le rendu d'une valeur dynamique, on fait appel à une expression en utilisant @

```
int Size = 8;
bool HeaderVisible = true;
string Text = "Text to display";
private int DoubleSize()
{
    return Size * 2;
}
```

## Razor view

```
<input size=@Size/>
<input size=@DoubleSize()/>
<MyHeader Text=@HeaderText
Visible=@HeaderVisible/>
```

## HTML

```
<input size="8"/>
<input size="16"/>
<h1>Value of variable</h1>
```

- Cas des expressions inférées : Blazor va contrôler que la valeur passée correspond au type attendu. Si c'est le cas il l'infère.

<MyComponent Visible="true"/>

Littérale de type string, inférée comme expression @true.

# LES DIRECTIVES

- Se sont des macros intégrés qui vont modifier le code C# transpilé, généré par les balises Razor
- Elles commencent par un @  
`<button @onclick="DoSomething">`
- L'ajout de @ devant la valeur est inutile, sauf pour transmettre une string (échappement), exemple pour les lambda :  
`@onclick=@(args => Debug.WriteLine("Clicked"))`
- Il n'est pas possible de créer ces propres directives pour le moment
- Exemple de directive pour les fichiers razors
  - @code, @page, @layout, @paramtype, @inject, etc...
- Exemple de directive pour les composants
  - @ref, @bind, @attributes, etc ..

# LES DIRECTIVES

```
// Razor mark-up with @ref directive
<h1 @onclick=H1Clicked>Hello, world!</h1>
@code {
    public void H1Clicked(MouseEventArgs args)
    {
        System.Diagnostics.Debug.WriteLine("H1 clicked");
    }
}
// Transpiled C#
public partial class Index : ComponentBase
{
    protected override void BuildRenderTree(RenderTreeBuilder __builder)
    {
        __builder.OpenElement(0, "h1");
        __builder.AddAttribute(1, "onclick",
            EventCallback.Factory.Create<MouseEventArgs>(this, H1Clicked));
        __builder.AddContent(2, "Hello, world!");
        __builder.CloseElement();
    }
}
```



## LES EVENTCALLBACK<T>

- Classe spéciale de Blazor qui peut être exposée comme paramètre
- Notifie le consommateur du composant que quelque chose d'intéressant s'est produit.

[Parameter]

```
public EventCallback<int> OnMultipleOfThree { get; set; }
```

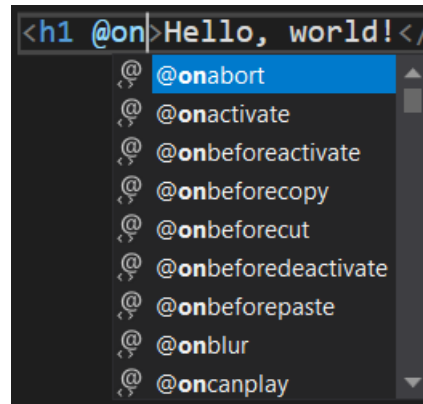
- Événement asynchrone (retourne une tâche)
- Pour faire appel à cet événement, il faut y souscrire

```
await OnMultipleOfThree.InvokeAsync(currentCount);
```

- Compatible avec les balises Razor
- Détecte automatiquement les changements d'état (exécution de StateHasChanged())

# LES ÉVÈNEMENTS DU DOM

- Il est possible d'assigner n'importe quels événements standard JavaScript à l'aide des directives Razor



- ⚠ Si on travaille en Blazor serveur, l'utilisation d'événement déclenché souvent (comme « onmousemove ») peuvent provoquer des ralentissements dû fait de leur envoi vers le serveur

## TWO-WAY BINDING

- Permet de notifier au parent du changement de valeur d'une propriété transmise par paramètre
- Nécessite l'utilisation de la directive `@bind-` et d'un EventCallback
- Le nom de l'EventCallback doit respecter la convention `SomeProperty => SomePropertyChanged`

- Dans le parent

```
<MyComponent @bind-CurrentValue="currentCount" />
```

- Dans le enfant

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

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

```
private async Task Increase()
```

```
{
```

```
    CurrentValue++;
```

```
    await CurrentValueChanged.InvokeAsync(CurrentValue);
```

```
}
```

# LES DIRECTIVES ATTRIBUT

- Directives liées aux attributs HTML
- Prennent la forme @directive:attribute
- Information additionnelle fourni à une directive

```
<input type="submit" @onclick:preventdefault>
```

- Standard two-way binding :

```
<input @bind-value=Name />
```

- Détection immédiate de changement

```
<input @bind-value=Name @bind-value:event="onchange" />
```

```
<input @bind-value=Name @bind-value:event="oninput" />
```

- Format spécifique

```
<input @bind-value=Date @bind-value:format="yyyy-MM-dd" />
```

- Culture spécifique

```
<input @bind-value=BankBalance @bind-value:culture=Turkish />
```

# LES PROPRIÉTÉS EN CASCADE

- Il est possible de transmettre une propriété à toute la liste des composant enfant en utilisant les propriétés en cascade
- Propriété en cascade par **nom**

- Dans le parent

```
<CascadingValue Name="FirstOption" Value=@Value>  
    <MyComponent />  
</CascadingValue>
```

- Dans les enfants

```
[CascadingParameter(Name = "FirstOption")]  
private bool FirstOption { get; set; }
```

# LES PROPRIÉTÉS EN CASCADE

- Propriété en cascade par **type**

- Dans le parent

```
<CascadingValue Value=@true>  
    <MyComponent />  
</CascadingValue>
```

- Dans les enfants

```
[CascadingParameter]  
private bool Property { get; set; }
```

# LES PROPRIÉTÉS EN CASCADE

- Il est possible de surcharger une propriété en cascade

```
<CascadingValue Name="CascadedValue" Value=@CascadedValue>
  <CascadingValue Name="ValueToOverride" Value=@OuterValue>
    <h2>First level</h2>
    <ViewSomeValue />

    <CascadingValue Name="ValueToOverride" Value=@InnerValue>
      <h2>Second level</h2>
      <ViewSomeValue />
    </CascadingValue>

    <h2>Back to first level</h2>
    <ViewSomeValue />
  </CascadingValue>
</CascadingValue>
```

## First level

Values are CascadedValue / Outer value

## Second level

Values are CascadedValue / Inner value

## Back to first level

Values are CascadedValue / Outer value

## ATTRIBUTE SPLATTING

- Création dynamique d'attribut HTML par du code Razor
- Nécessite l'utilisation d'un Dictionary<string,object>

```
<div @attributes=MyCodeGeneratedAttributes />
```

```
@code
```

```
{
```

```
Dictionary<string, object> MyCodeGeneratedAttributes;
```

```
protected override void OnInitialized()
```

```
{
```

```
    MyCodeGeneratedAttributes = new Dictionary<string, object>();
```

```
    for (int index = 1; index <= 5; index++)
```

```
    {
```

```
        MyCodeGeneratedAttributes["attribute_" + index] = index;
```

```
    }
```

```
}
```

```
}
```

```
<div attribute_1="1" attribute_2="2" attribute_3="3" attribute_4="4"  
attribute_5="5"></div>
```



## ATTRIBUTE SPLATTING

- Cas particulier readonly et disabled, leur présence suffit pour les rendre effectifs
  - Même si on leur attribue false, ils seront quand même activés  
`<input readonly="false" disabled="false" />`
- Leur comportement dans les composants Razor est différent
- En effet

`readonly=@IsReadOnly`

`disabled=@IsDisabled`

Lié à des propriétés de type booléen vont faire qu'ils soient rendu ou pas en HTML par le moteur Razor

# LA CAPTURE DE PARAMÈTRE

- Il est possible de capturer des valeurs d'attribut sans paramètre dans un Dictionary<string,object>

- Appel du composant

```
<MyCustomImage src="https://url.me/6.jpg" width=64 height=64 />
```

- Dans le composant

```
[Parameter(CaptureUnmatchedValues = true)]
```

```
public Dictionary<string, object> AllOtherAttributes { get; set; }
```

- Ce dictionnaire servira au moteur Razor pour piocher les attributs et leur valeur dans le code HTML

- Utilisation du dictionnaire

```

```

## LA SURCHARGE D'ATTRIBUTS

- En déclarant des valeurs par défaut pour certain (ou tous) les attributs, il est possible des les surcharger à l'aide de la méthode de capteurs des paramètres en les passant au composant enfant

- Dans le composant parent:

```
<ChildComponent first="consumer-value-1" second="consumer-value-2" />
```

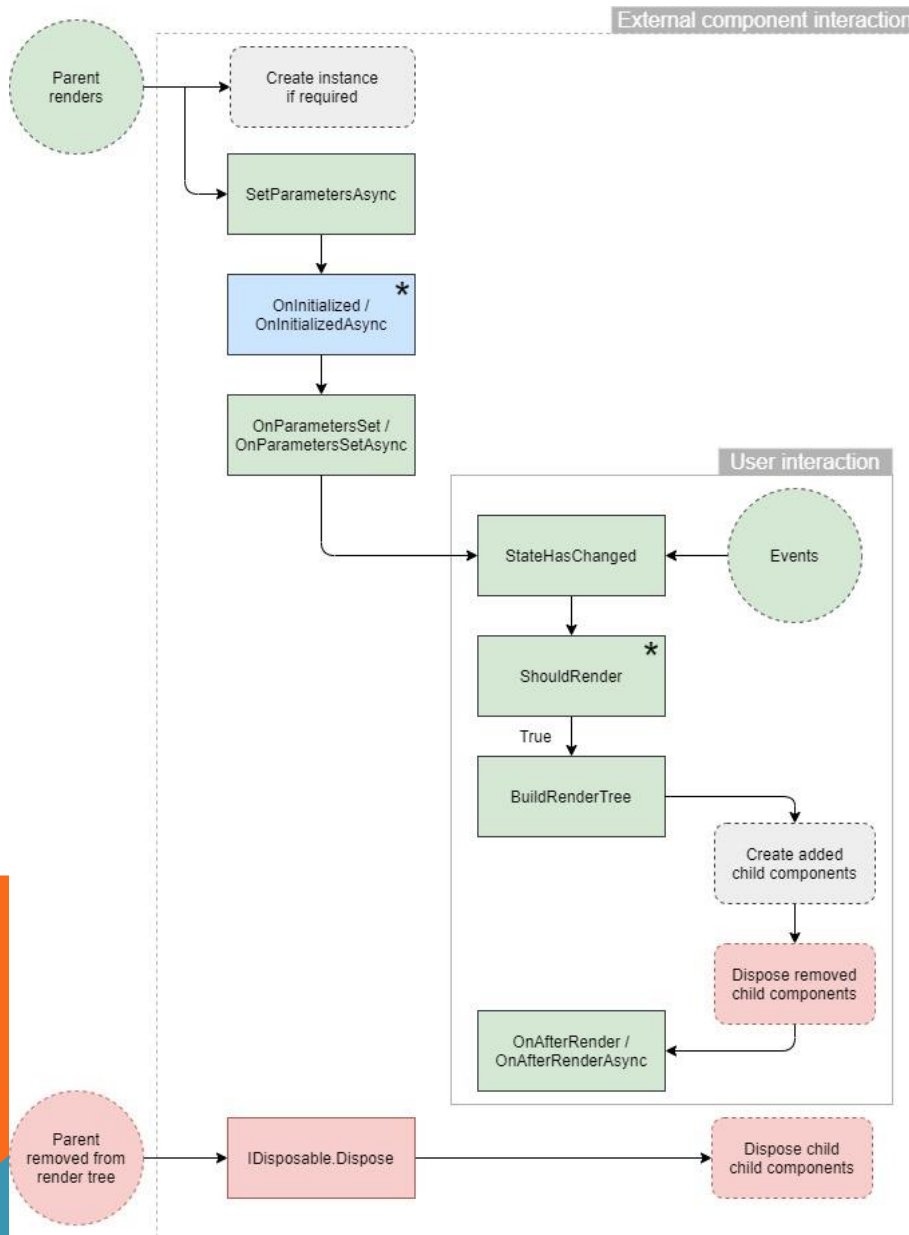
- Dans le composant enfant

```
<div first="1" second="2" third="3" fourth="4"  
  @attributes=AllOtherAttributes>  
</div>
```

- Résultat HTML

```
<div first="consumer-value-1" second="consumer-value-2"  
  third="3" fourth="4">  
</div>
```

# CYCLE DE VIE DES COMPOSANTS



- SetParametersAsync
- OnInitialized / OnInitializedAsync
- OnParametersSet / OnParametersSetAsync
- StateHasChanged
- ShouldRender
- BuildRenderTree
- OnAfterRender / OnAfterRenderAsync
- Dispose

## **BLAZOR SERVER ET MULTI-THREADING**

- Comme plusieurs threads sont disponibles dans Blazor Server, il est possible que certain composant ait du code exécuté par différents threads
- Valable pour toutes les méthodes renvoyant une tâche
- Utilisation de `InvokeAsync()` pour exécuter le code de manière thread-safe (appelle du Dispatcher)
- Ex : `StateHasChanged()`, n'autorise pas un accès multi-threader au processus de rendu. Donc si 2 threads veulent accéder au rendu, on va avoir une erreur  
=> Nécessité d'aller le dispatcher avec `InvokeAsync()`

QUESTIONS ? REMARQUES ?



# DÉMONSTRATION



# MODULE#05

LE RENDER TREE





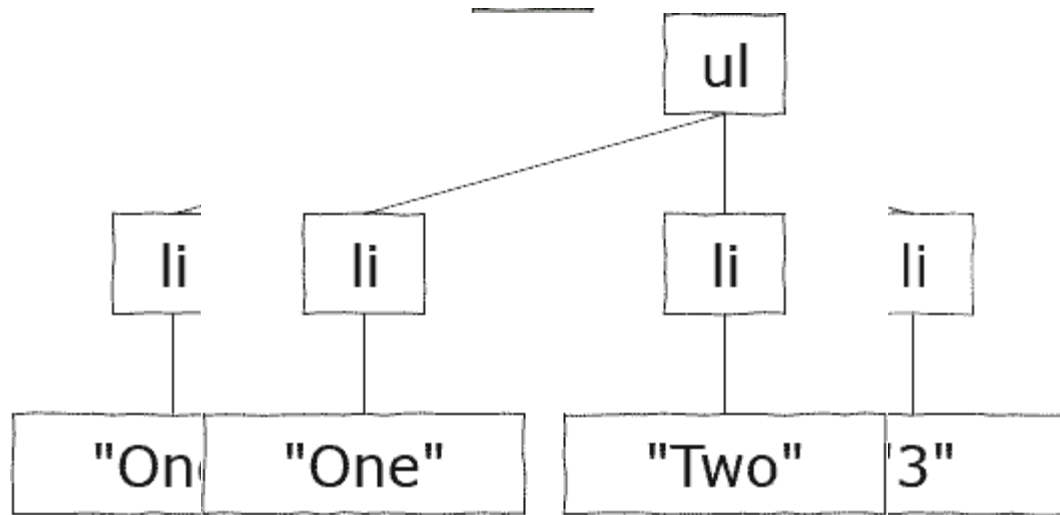
## RENDER TREES

- Lors d'un rendu de composant, le navigateur :
  - Dessine les éléments définis par le HTML
  - Calcul l'endroit où il doit être rendu en fonction de la taille de la page
  - A chaque changement d'une propriété type width, padding, height, margin, etc, il recommence les calculs
  - => Mise à jour du DOM peut être très consommateur de CPU
- Utilisation d'un DOM virtuel et de la méthode incrémentielle
  - DOM virtuel = représentation en mémoire des éléments HTML
  - Création d'un arbre de rendu.
  - Appel de BuildRenderTree() pour créer le DOM Virtuel

```
protected override void BuildRenderTree(RenderTreeBuilder builder)
{
    builder.AddMarkupContent(0, "<h1>Hello, world!</h1>");
    builder.OpenComponent<SurveyPrompt>(1);
    builder.AddAttribute(2, "Title", "Blazor App?");
    builder.CloseComponent();
}
```

## DOM INCRÉMENTAL

- Technique qui minimise la quantité de travail nécessaire pour mettre à jour les éléments de la vue



## LA DIRECTIVE @KEY

- @Key : signifie à Blazor de traquer les éléments qui ont été réarrangé
- Refait le rendu complet de l'élément traquer par @key, lorsqu'il a changé

```
@foreach (Person person in People)
{
    <li class="card" @key="person">
        
        <div class="card-body">
            <h5 class="card-title">
                @person.GivenName @person.FamilyName
            </h5>
            <p class="card-text">
                @person.GivenName @person.FamilyName has the id @person.ID
            </p>
        </div>
    </li>
}
```

QUESTIONS? REMARQUES?



# DÉMONSTRATION



# MODULE#06

LES TEMPLATES DE COMPOSANT



## UTILISATION DES RENDERFRAGMENTS

- Il est possible de transmettre un bloque de code HTML ou un autre composant à un composant Blazor
- On déclare un paramètre de type RenderFragment dans le composant

[Parameter]

```
public RenderFragment ChildContent { get; set; }
```

- On utilise le RenderFragment comme ceci :

```
<div class="col-12 card card-body">
```

```
    @ChildContent
```

```
</div>
```

- Dans le composant parent, on transmet le code au RenderFragment de l'enfant comme ci-dessous

```
<MyComponent>Hello world!</MyComponent>
```

## UTILISATION DES RENDERFRAGMENTS

- Si il n'y a qu'un RenderFragment et qu'il est nommé ChildContent, il suffit de mettre le code entre les balises du composant
- Si ils sont plusieurs ou qu'il est nommé autrement, il faut procéder comme suit

```
<MyComponent>  
  <ContentToAdd>  
    Hello world!  
  </ContentToAdd>  
</MyComponent>
```

- On va faire appelle à ce code dans le composant en utilisant @ContentToAdd

```
<div>  
  @ContentToAdd  
</div>
```



# UTILISATION DES RENDERFRAGMENTS

- Il est possible de transmettre des données à un RenderFragment
  - Utilisation du RenderFragment<T>

```
public RenderFragment<MyObject> ContentToAdd { get; set; }
```

- On va pouvoir passer des données à ce RenderFragment

```
@foreach (var obj in MyObjects)
{
    <div>
        @ContentToAdd(obj)
    </div>
}
```

- Dans le parent on va définir le code du fragment comme ceci

```
<MyComponent>
    <ContentToAdd>
        <div>@context.Text </div>
    </ContentToAdd>
</MyComponent>
```

# CREATION DE COMPOSANT GÉNÉRIQUE

- Il est possible de créer des composants générique en utilisant la directive @typeparam

```
@typeparam TItem
```

```
@code
```

```
{
```

```
    [Parameter]
```

```
    public IEnumerable<TItem> Data { get; set; }
```

```
}
```



QUESTIONS ? REMARQUES ?



# DÉMONSTRATION




# MODULE#04

LE ROUTING



# LA NAVIGATION DANS BLAZOR

- Ne navigue pas au sens WWW
- Lors d'une navigation, Blazor réécrit l'url du navigateur et fait le rendu du contenu demandé
-  Lors d'une navigation vers le même composant, il n'est pas détruit et OnInitialized ne sera pas appelé.
  - => Vu comme une mise à jour des composants
- La directive **@page** définit la route pour un composant
  - Traduit comme un RouteAttribut
- Blazor scan les classes pour récupérer les RouteAttribut et créer ces routes en fonction de ça
  - = Route Discovery (effectué automatiquement)

# ROUTE DISCOVERY

- Défini dans App.razor

```
<Router AppAssembly="@typeof(Program).Assembly">  
</Router>
```

- Scan toutes les classes liées à l'Assembly
- Pour chaque RouteAttribut trouvé
  - Créé une URL
  - Garde un lien entre l'URL et le composant
- Un composant sans @page ne peut pas être appelé par URL
- Un composant avec de multiples @page peut être appelé par chacune des URLs défini de cette manière

```
@page "/"  
@page "/greeting"  
@page "/HelloWorld"  
@page "/hello-world"  
<h1>Hello, world!</h1>
```

# PARAMÈTRE DES ROUTES

- Les paramètres de route sont définis par { }  
`@page "/customer/{CustomerId}"`
- Il suffit de déclarer un paramètre de même nom dans le composant pour récupérer cette valeur  
`[Parameter]  
public string CustomerId { get; set; }`
- Il est possible de restreindre le type du paramètre transmis  
`@page "/purchase-order/{OrderNumber:int}"`



# PARAMÈTRE DES ROUTES

Contrainte	Type .NET	Valide	Invalide
:bool	System.Boolean	<ul style="list-style-type: none"><li>•true</li><li>•false</li></ul>	<ul style="list-style-type: none"><li>•1</li><li>•Hello</li></ul>
:datetime	System.DateTime	<ul style="list-style-type: none"><li>•2001-01-01</li><li>•02-29-2000</li></ul>	<ul style="list-style-type: none"><li>•29-02-2000</li></ul>
:decimal	System.Decimal	<ul style="list-style-type: none"><li>•2.34</li><li>•0.234</li></ul>	<ul style="list-style-type: none"><li>•2,34</li><li>•0.ṙ3 Ȳ</li></ul>
:double	System.Double	<ul style="list-style-type: none"><li>•2.34</li><li>•0.234</li></ul>	<ul style="list-style-type: none"><li>•2,34</li><li>•0.ṙ3 Ȳ</li></ul>
:float	System.Single	<ul style="list-style-type: none"><li>•2.34</li><li>•0.234</li></ul>	<ul style="list-style-type: none"><li>•2,34</li><li>•0.ṙ3 Ȳ</li></ul>
:guid	System.Guid	<ul style="list-style-type: none"><li>•99303dc9-8c76-42d9-9430-de3ee1ac25d0</li></ul>	<ul style="list-style-type: none"><li>•{99303dc9-8c76-42d9-9430-de3ee1ac25d0}</li></ul>
:int	System.Int32	<ul style="list-style-type: none"><li>•-1</li><li>•42</li><li>•299792458</li></ul>	<ul style="list-style-type: none"><li>•12.34</li><li>•ṙ3</li></ul>
:long	System.Int64	<ul style="list-style-type: none"><li>•-1</li><li>•42</li><li>•299792458</li></ul>	<ul style="list-style-type: none"><li>•12.34</li><li>•ṙ3</li></ul>

# PARAMÈTRE DES ROUTES

- Blazor ne prend pas en charge explicitement les paramètres optionnels.
- Toutefois, un comportement similaire est obtainable en définissant plusieurs @page

```
@page "/counter"
```

```
@page "/counter/{CurrentCount:int}"
```

- Il est possible de modifier à la valeur par défaut du paramètre optionnel en utilisant la OnParameterSetAsync()

```
public async override Task SetParametersAsync(ParameterView parameters)
{
    await base.SetParametersAsync(parameters);
    CurrentCount = CurrentCount ?? 1;
}
```

## 404 – NOT FOUND

- Il est possible d'indiquer à Blazor la page à afficher lorsqu'il ne trouve pas l'url
- Défini dans App.razor

```
<Router AppAssembly="@typeof(Program).Assembly">
    <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>
```

# NAVIGATION VIA HTML

- Pour navigué via HTML il est possible de:
  - Créé un lien `<a>`  
`<a href="/Counter">This works just fine</a>`
  - D'utilisant le composant NavLink  
`<NavLink class="nav-link" href="counter">`  
 `<span class="oi oi-home" aria-hidden="true"></span> Counter`  
`</NavLink>`
- Le paramètre `ActiveClass` du `NavLink` indique le rendu du HTML lorsque l'url correspond à celle du navigateur
- Le paramètre `Match` indique le type de comparaison entre l'url du navigateur et celle du `NavLink`
  - `NavLinkMatch.All` => url identique
  - `NavLinkMatch.Prefix` = > les paramètres peuvent différés

## NAVIGATION VIA CODE

- Pour navigué via le code c#, il est existe un NavigationManager  
`@inject NavigationManager Navigation`
- Utilisation de la méthode NavigateTo() pour naviguer
  - Déclenche un évènement LocationChanged
  - Paramètre ForceLoad de cette méthode indique à Blazor de bypasser son système de route et fait une requête vers le serveur

```
NavigationManager.NavigateTo("/counter/" + newCount,  
forceLoad);
```

# **DÉTECTION DES ÉVÈNEMENTS DE NAVIGATION**

- A chaque navigation un évènement `LocationChanged` est levé
  - Il contient une instance de `LocationChangedEventArgs`
  - Parmi ces arguments, `IsNavigationIntercepted` indique si la navigation a été initialisé depuis le code ou depuis un lien HTML
    - `False` => provient du code
    - `True` => provient de HTML (ou Javascript)
- Il est possible d'observer les évènements de navigation

# DÉTECTION DES ÉVÈNEMENTS DE NAVIGATION

```
protected override void OnInitialized()
{
    // Subscribe to the event
    NavigationManager.LocationChanged += LocationChanged;
    base.OnInitialized();
}
void LocationChanged(object sender,
LocationChangedEventArgs e)
{
    string navigationMethod = e.IsNavigationIntercepted ?
"HTML" : "code";
    System.Diagnostics.Debug.WriteLine($"Notified of
navigation via {navigationMethod} to {e.Location}");
}
void IDisposable.Dispose()
{
    // Unsubscribe from the event when our component is
disposed
    NavigationManager.LocationChanged -= LocationChanged;
}
```

QUESTIONS ? REMARQUES ?





# DÉMONSTRATION



# MODULE#07

LES FORMULAIRES



# EDITFORM

- EditForm = composant Blazor pour les formulaires
  - Possède un paramètre Model => c'est le contexte du composant

```
<EditForm Model=@Person>
```

```
    <input type="submit" value="Submit" class="btn btn-primary" />
```

```
</EditForm>
```

- Lorsque l'on va cliquer sur le bouton de soumission on la déclencher le OnSubmit de l'EditForm

```
<EditForm Model=@Person OnSubmit=@FormSubmitted>
```

```
    <input type="submit" value="Submit" class="btn btn-primary" />
```

```
</EditForm>
```

```
void FormSubmitted()
```

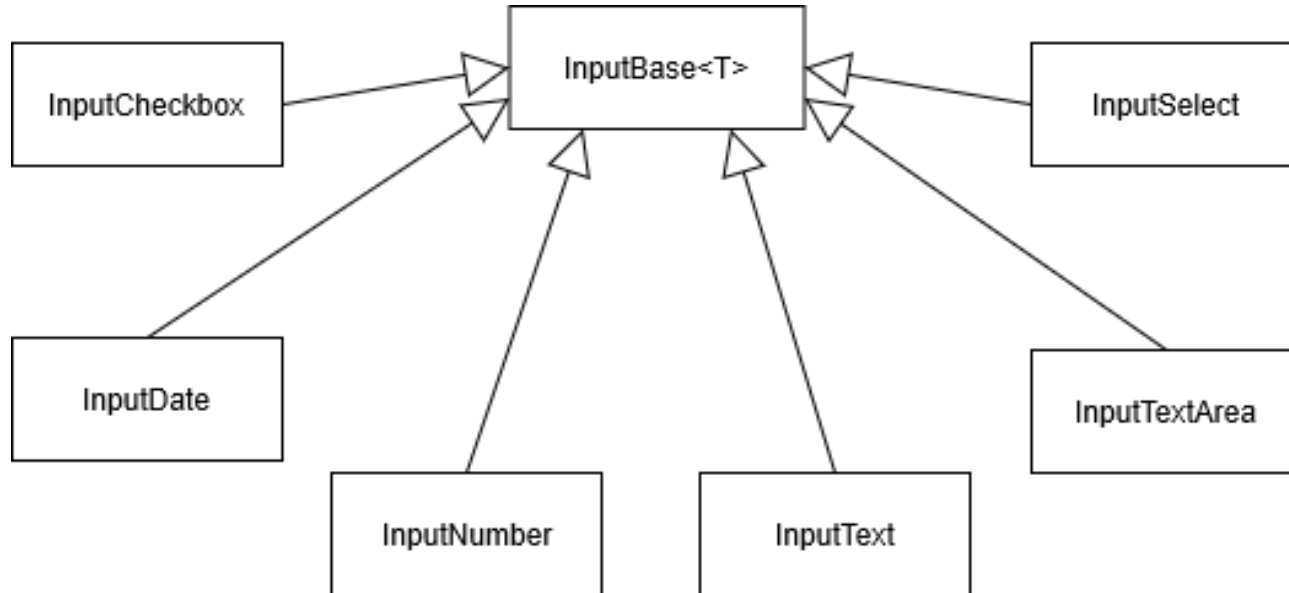
```
{
```

```
    Status = "Form submitted";
```

```
}
```

## DONNÉES DU FORMULAIRE

- EditForm est traduit en <form> HTML
  - Possible d'utiliser les balises de formulaire standard (input, select, etc)
  - Il existe un set de composant Blazor dédié au formulaire



- Il est possible de créer ces propres composants de formulaire en les faisant hériter de `InputBase<T>`

# LA VALIDATION

- Le composant de base pour la validation des formulaires est `DataAnnotationValidator`
- Fonctionne avec les attributs de validation
- `ValidationSummary` affiche la liste des messages d'erreur
- `ValidationMessage` affiche le message d'erreur pour un composant spécifique

```
<EditForm Model=@Personne>
    <DataAnnotationsValidator/>
    <ValidationSummary/>
    <InputText @bind-Value="Personne.Name"></InputText>
    <ValidationMessage For=@(()=>Personne.Name)/>
    <input type="submit" value="Submit" class="btn btn-primary" />
</EditForm>
```

# LES ÉVÈNEMENTS DE SOUMISSION

- En Blazor, il existe 3 évènements qui peuvent être appelé lors de la soumission
  - OnValidSubmit => lorsque la validation est Ok
  - OnInvalidSubmit => lorsque la validation est Non Ok
  - OnSubmit => quelque soit le statut de validation
- Chacun des 3 reçoit un EditContext en paramètre
- Avec OnSubmit, il est possible de vérifier l'état de la validation à l'aide de editContext.Validate()

```
<EditForm Model=@Person OnSubmit=@FormSubmitted>
    <DataAnnotationsValidator />
    <input type="submit" class="btn btn-primary" value="Save" />
</EditForm>

@code {
    void FormSubmitted(EditContext editContext)
    { bool formIsValid = editContext.Validate(); }
}
```

QUESTIONS ? REMARQUES ?



# DÉMONSTRATION





# MODULE#08

L'INTEROP JS



## **JAVASCRIPT INTEROP**

- Pour toutes les fonctionnalités du navigateur que ne proposent pas Blazor, on peut se servir de Javascript comme intermédiaire.
- Quelques bonnes pratiques pour utiliser Javascript avec Blazor:
  - Ne pas utiliser InteropJS durant la phase de pré-rendu du serveur
  - Ne pas utiliser des objets ElementReference trop tôt
  - Attention aux fuites de mémoire, penser à libérer les ressources
  - Ne pas appeler de méthode d'objet .NET détruits
  - Ne pas appeler de méthode .NET avant que Blazor ne soit initialisé

# **DÉMARRAGE DE JAVASCRIPT**

- Durant le démarrage de Blazor, le document HTML est créé avant l'initialisation de Blazor
  - Les méthodes Javascript sont disponibles immédiatement
  - Le code auto-exécuté est exécuté avant que Blazor ne soit disponible
  - Il n'est pas possible d'appeler du code .NET depuis du code exécuté par Javascript.  
=> Initialisé le traitement en démarrant du code .NET
- Le code Javascript doit être déclaré dans `_Host.cshtml` en Blazor Server et dans `Home.html` dans Blazor WASM

# DÉMARRAGE DE JAVASCRIPT

- Pour appeler une fonction Javascript depuis Blazor, on a besoin du service IJSRuntime

**@inject IJSRuntime JSRuntime**

- A l'aide de JSRuntime, on peut faire appel à une méthode Javascript et lui transmettre des paramètres

```
private async Task ButtonClicked()  
{  
    await JSRuntime.InvokeVoidAsync("alert",  
"Hello world");  
}
```

- Il est possible de passer des objets complexes en paramètre
  - Ils sont sérialisé en JSON puis désérialisé en Javascript
- `InvokeAsync<TValue>` permet de récupérer les valeurs de retour de la méthode appelée

# RÉFÉRENCES HTML

- Il est possible de récupérer la référence à un élément HTML (pour le transmettre à une fonction JS par exemple)
- Utilisation de la directive `@ref` dans l'élément HTML
- Déclaration d'un `ElementReference` dans le code qui sera utilisé avec `@ref`

```
<h1 @ref=MyElementReference>Hello, world!</h1>  
@code {  
    ElementReference MyElementReference;  
}
```

# APPELER DU .NET À PARTIR DE JAVASCRIPT

- Pour appeler une méthode .NET à partir de JS, il faut
  - Que la méthode soit décorer avec JsInvokableAttribute
  - Qu'elle soit public
  - Que ces paramètres soient sérialisable en Json
  - Que le type de retour soit sérialisable en Json ou void ou Task ou Task<T> où T est sérialisable en Json
  - Si le paramètre « identifier » est défini, il doit être unique
- Il faut commencer par transmettre une instance de l'objet .NET à Javascript

```
var dotNetReference = DotNetObjectReference.Create(this);
```

- On peut alors se servir de cette objet pour appeler une de ces méthodes qui répons au condition si dessus

```
var func = function (dotNetObject) {  
    let text = Math.random() * 1000;  
    dotNetObject.invokeMethodAsync('AddText', text.toString());  
};
```

## CYCLE DE VIE ET FUITE MÉMOIRE

- Lorsque l'on passe un objet dotNetReference à Javascript, Blazor génère un ID unique pour l'objet (int pour wasn, guid pour server) et stock une entrée de recherche pour l'objet dans JSRuntime
- Si l'objet n'est pas détruit correctement, il reste utiliser et cela va provoquer une fuite de mémoire
- Le composant créant l'objet dotNetReference doit en conserver une référence et doit implémenter IDisposable
- On libérera les ressources lié au dotNetReference dans la méthode Dispose() du composant
- **Attention à ne plus appeler les méthodes de l'objet une fois disposer !**
- Si la fonction Javascript consommant l'objet dotNetReference est récurrente, il est nécessaire d'implémenter une fonction JS supplémentaire pour arrêter la récurrence. Elle sera appelée depuis la méthode Dispose du composant.

## TYPE DE RETOUR

- Le .NET étant fortement typé, lorsque l'on appelle une méthode .NET depuis JS, il faut faire attention au type des paramètres

JavaScript type	.NET type
boolean	System.Boolean
string	System.String
number	System.Float / System.Decimal System.Int32 (etc) if no decimal value
Date	System.DateTime or System.String

- Dans le cas d'un énum, JS doit renvoyer la représentation numérique de celui-ci.
- Si l'énum est décoré avec [Json.Serialization.JsonConverter], il devient alors possible de lui transmettre une représentation sous forme de string



## **APPEL DE MÉTHODE .NET STATIC**

- Il est aussi possible d'appeler des méthodes .NET static depuis JS
- Quelques règles à respecter
  - La classe possédant la méthode doit être public
  - La méthode doit être public
  - Elle doit être static
  - Le type de retour doit être sérialisable en Json ou une Task ou une Task<T> dont T est sérialisable en Json
  - Tous les paramètres doivent être sérialisable en Json
  - La méthode doit être décoré avec JSInvokable

# EXEMPLE D'APPEL DE MÉTHODE .NET STATIC

```
public static class JavaScriptConfiguration
{
    private static JavaScriptSettings Settings;
    internal static void SetSettings(JavascriptSettings
settings)
    { Settings = settings; }
    public static JavaScriptSettings GetSettings() => Settings;
}

setTimeout(async function () {
    const settings = await DotNet.invokeMethodAsync("CallingStaticDotNetMethods",
"GetSettings");
    alert('API key: ' + settings.someApiKey);
}, 1000);
```

QUESTIONS ? REMARQUES ?



# DÉMONSTRATION



# MODULE#10

SÉCURITÉ AVEC BLAZOR



# **AUTHENTIFICATION BLAZOR**

- Basé sur les mécanismes d'authentification d'ASP.NET Core
  - Peut implémenté le Framework Identity (template)
  - Peut utiliser d'autre mode d'authentification
    - Google
    - Azure ADAuthentication
    - OAuth
    - 2 facteurs,
    - etc...
- Implémentation dépend de la version de Blazor utilisé

# AUTHENTIFICATION BLAZOR

- Blazor WebAssembly
  - Authentification basé sur les cookies
    - JWT
  - Authentification API – Aller retour requête HTTP
- Blazor Server
  - Cookie utilisé uniquement pour persisté l'utilisateur connecté.
  - AuthenticationStateProvider
    - Indique l'état de l'authentification
    - va être utiliser pour lire les données du `HttpContext.User`
  - Echange WS Secured

## SERVICES ASP.NET CORE AUTHENTICATION

- Pour permettre authentification, nécessite ajout 2 middleware dans le pipeline de requête
- Dans **Program.cs**
  - Coté Server dans Blazor WASM
  - Dans l'unique **Program.cs** dans Blazor Server

```
app.UseAuthentication();  
app.UseAuthorization();
```

- A ajouté avant les middlewares de configuration des endpoints



# AUTHENTICATIONSTATEPROVIDER SERVICE

- Présent en Blazor WASM et Blazor Server
- Service derrière la mécanique des composant `AuthorizeView` et `CascadingAuthenticationState`
- Notifie automatiquement l'application si l'état de l'authentification à changer
- Fourni les données du `ClaimsPrincipal` de l'utilisateur actuel
- Ne pas utiliser directement, utiliser
  - `AuthorizeView`
  - `Task<AuthenticationState>`

# EXPOSER L'ÉTAT D'AUTHENTIFICATION

- Exposer par un paramètre en cascade

```
<button @onclick="LogUsername">Log username</button>  
<p>@authMessage</p>
```

```
@code {  
    [CascadingParameter] private Task<AuthenticationState> authStateTask  
    { get; set; }  
  
    private string authMessage;  
  
    private async Task LogUsername()  
    {  
        var authState = await authStateTask;  
        var user = authState.User;  
  
        if (user.Identity.IsAuthenticated)  
            authMessage = $"{user.Identity.Name} is authenticated."  
        else  
            authMessage = "The user is NOT authenticated."  
    }  
}
```

# EXPOSER L'ÉTAT D'AUTHENTIFICATION

- Pour Blazor WASM, nécessite 2 services pour fonctionner côté Client

```
builder.Services.AddOptions();  
builder.Services.AddAuthorizationCore();
```

- Services déjà présents dans Blazor Server

# L'AUTORISATION

- L'autorisation d'accès sera donnée à l'utilisateur en fonction de:
  - Si il est authentifié
  - Si il a le rôle adéquat
  - Si il a un « claim »
  - Si une politique est remplie
- Concept commun à toutes les applications web ASP.NET Core (Blazor, MVC, Razor Pages)

## AUTHORIZEVIEW

- Affiche le contenu du composant si l'utilisateur est autorisé à voir le contenu
- Expose un `@context` de type `AuthenticateState` permettant d'accéder aux infos de l'utilisateur

```
<AuthorizeView>  
  <h1>Hello, @context.User.Identity.Name!</h1>  
  <p>You can only see this content if you're authenticated.</p>  
</AuthorizeView>
```

## AUTHORIZEVIEW

- Il est aussi possible de spécifier le contenu à afficher si l'utilisateur n'est pas autorisé

```
<AuthorizeView>
  <Authorized>
    <h1>Hello, @context.User.Identity.Name!</h1>
    <p>You can only see this content if you're
authorized.</p>
  </Authorized>
  <NotAuthorized>
    <h1>Authentication Failure!</h1>
    <p>You're not signed in.</p>
  </NotAuthorized>
</AuthorizeView>
```

# AUTHORIZEVIEW

- `AuthorizeView` peut vérifier si l'utilisateur possède un rôle

```
<AuthorizeView Roles="admin, superuser">
    <p>You can only see this if you're an admin or superuser.</p>
</AuthorizeView>
```

- Ou encore si il respecte une politique spécifique

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("LoggedInPolicy", policy => policy.RequireAuthenticatedUser());
});
```

```
<AuthorizeView Policy="LoggedInPolicy">
    <p>You can only see this if you satisfy the "content-editor" policy.</p>
</AuthorizeView>
```

# AUTHORIZEVIEW

- Par défaut `AuthorizeView` n'affiche rien tant que l'authentification est en cours
- Utilisation de `Authorizing` pour afficher du contenu lors de l'authentification

```
<AuthorizeView>
  <Authorized>
    <h1>Hello, @context.User.Identity.Name!</h1>
    <p>You can only see this content if you're
authenticated.</p>
  </Authorized>
  <Authorizing>
    <h1>Authentication in progress</h1>
    <p>You can only see this content while authentication
is in progress.</p>
  </Authorizing>
</AuthorizeView>
```



## RESTRICTION PAR ATTRIBUT

- Il est possible de restreindre l'accès à des composants complets à l'aide de l'attribut `Authorize`

```
@attribute [Authorize]
```

```
@attribute [Authorize(Roles = "admin, superuser")]
```

```
@attribute [Authorize(Policy = "content-editor")]
```

# CONTENU CUSTOM NON AUTORISÉ

- Router conjugué avec `AuthorizeRouteView` permet d'afficher du contenu spécifique à la situation:
  - Utilisateur non identifié
  - Authentification en cours
  - Contenu non trouvé

```
<CascadingAuthenticationState>
  <Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
      <AuthorizeRouteView RouteData="@routeData"
        DefaultLayout="@typeof(MainLayout)">
        <NotAuthorized>
          <p>Sorry, You're not authorized to reach this page.</p>
        </NotAuthorized>
        <Authorizing>
          <h1>Authorization in progress</h1>
        </Authorizing>
      </AuthorizeRouteView>
    </Found>
    <NotFound>
      <LayoutView Layout="@typeof(MainLayout)">
        <p>Sorry, there's nothing at this address.</p>
      </LayoutView>
    </NotFound>
  </Router>
</CascadingAuthenticationState>
```

## AUTHENTICATIONSTATE DANS LE CODE C#

- L'application peut contrôler l'autorisation d'un utilisateur depuis son code
- Nécessite d'utiliser un paramètre en cascade de type `Task<AuthenticationState>`
- Obtention du `ClaimsPrincipal` de l'utilisateur

```
[CascadingParameter] private Task<AuthenticationState> authStateTask { get; set; }

private async Task DoSomething()
{
    ClaimsPrincipal user = (await authStateTask).User;

    if (user.Identity.IsAuthenticated)
        // Action.

    if (user.IsInRole("admin"))
        // Action.

    if ((await AuthorizationService.AuthorizeAsync(user, "content-editor"))
        .Succeeded)
        //Action
}
```

QUESTIONS ? REMARQUES ?



# DÉMONSTRATION



# MODULE#11

LE DÉPLOIEMENT BLAZOR



# DÉPLOIEMENT BLAZOR

- Blazor = ASP.NET Core
- Possède un server web intégré = Kestrel
- Possibilité de déploiement Multi-plateforme
  - Windows
  - Linux
  - MacOS
  - Docker
  - ...
- Possibilité de créer des builds contenant le framework ASP.NET correspondant = autonome

## PUBLIER DES APPLICATIONS BLAZOR

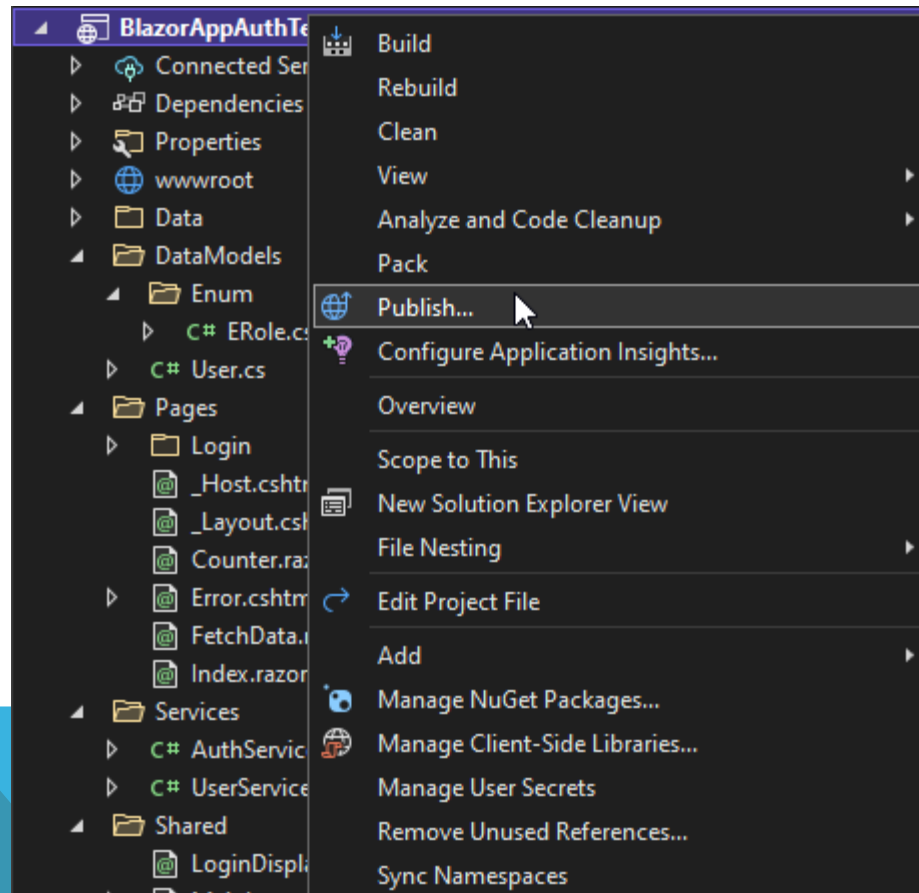
- En ligne de commande ASP.NET Core

```
dotnet publish [<PROJECT> | <SOLUTION>] [-a | --arch <ARCHITECTURE>]  
    [-c | --configuration <CONFIGURATION>]  
    [-f | --framework <FRAMEWORK>] [--force] [--interactive]  
    [--manifest <PATH_TO_MANIFEST_FILE>] [--no-build]  
    [--no-dependencies]  
    [--no-restore] [--nologo] [-o | --output <OUTPUT_DIRECTORY>]  
    [--os <OS>] [-r | --runtime <RUNTIME_IDENTIFIER>]  
    [--self-contained [true | false]] [--no-self-contained]  
    [-s | --source <SOURCE>] [-v | --verbosity <LEVEL>]  
    [--version-suffix <VERSION_SUFFIX>]  
dotnet publish -h | --help
```



# PUBLIER DES APPLICATIONS BLAZOR

- A l'aide de Visual Studio



# DÉMONSTRATION

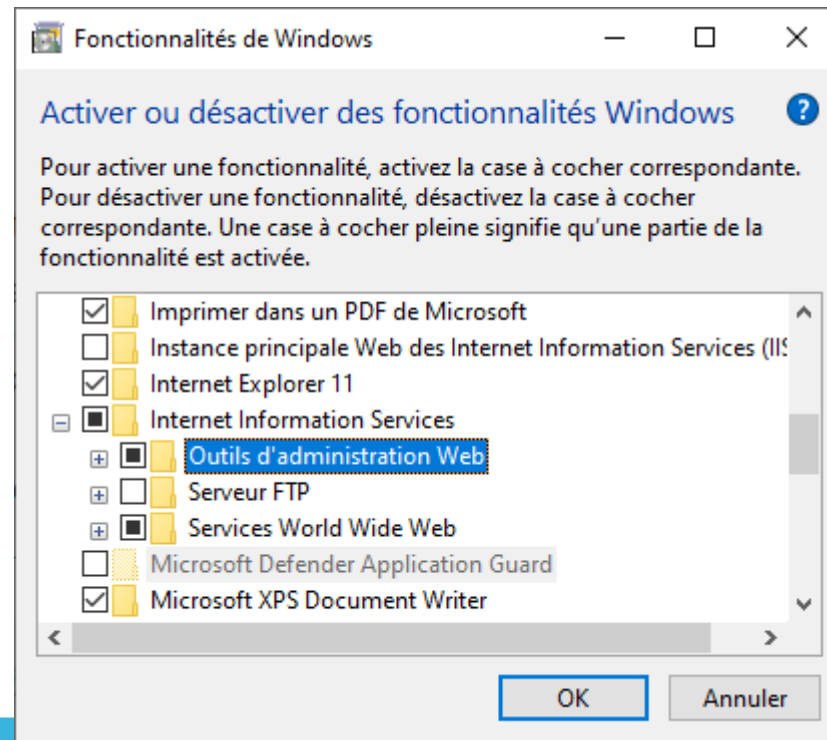


## **HÉBERGÉ UNE APPLICATION BLAZOR**

- Dans un pool d'application IIS (1 par app)
- Dans un service Windows
- Dans une Github Page
- Dans un daemon Linux
- Dans un conteneur Docker
- ...

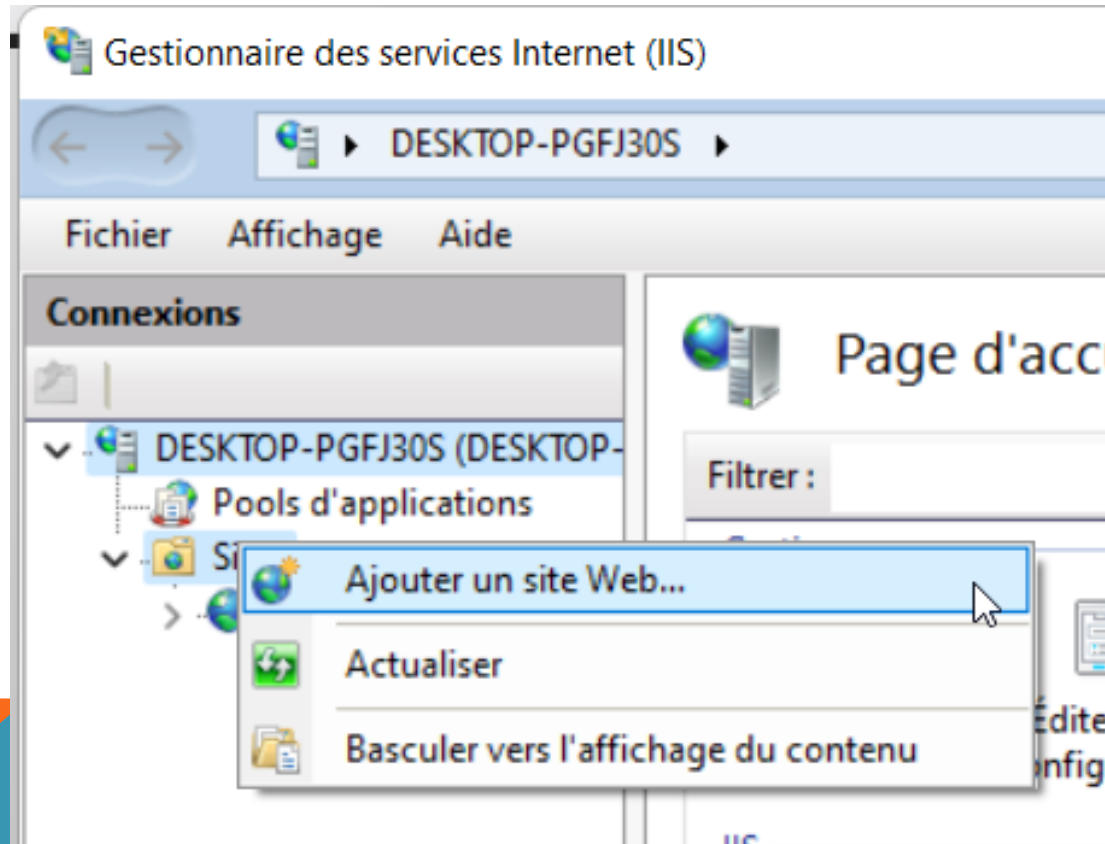
# HÉBERGEMENT AVEC IIS EXPRESS

- Activer IIS



# HÉBERGEMENT AVEC IIS EXPRESS

- Création du site web sous IIS



# HÉBERGEMENT AVEC IIS EXPRESS

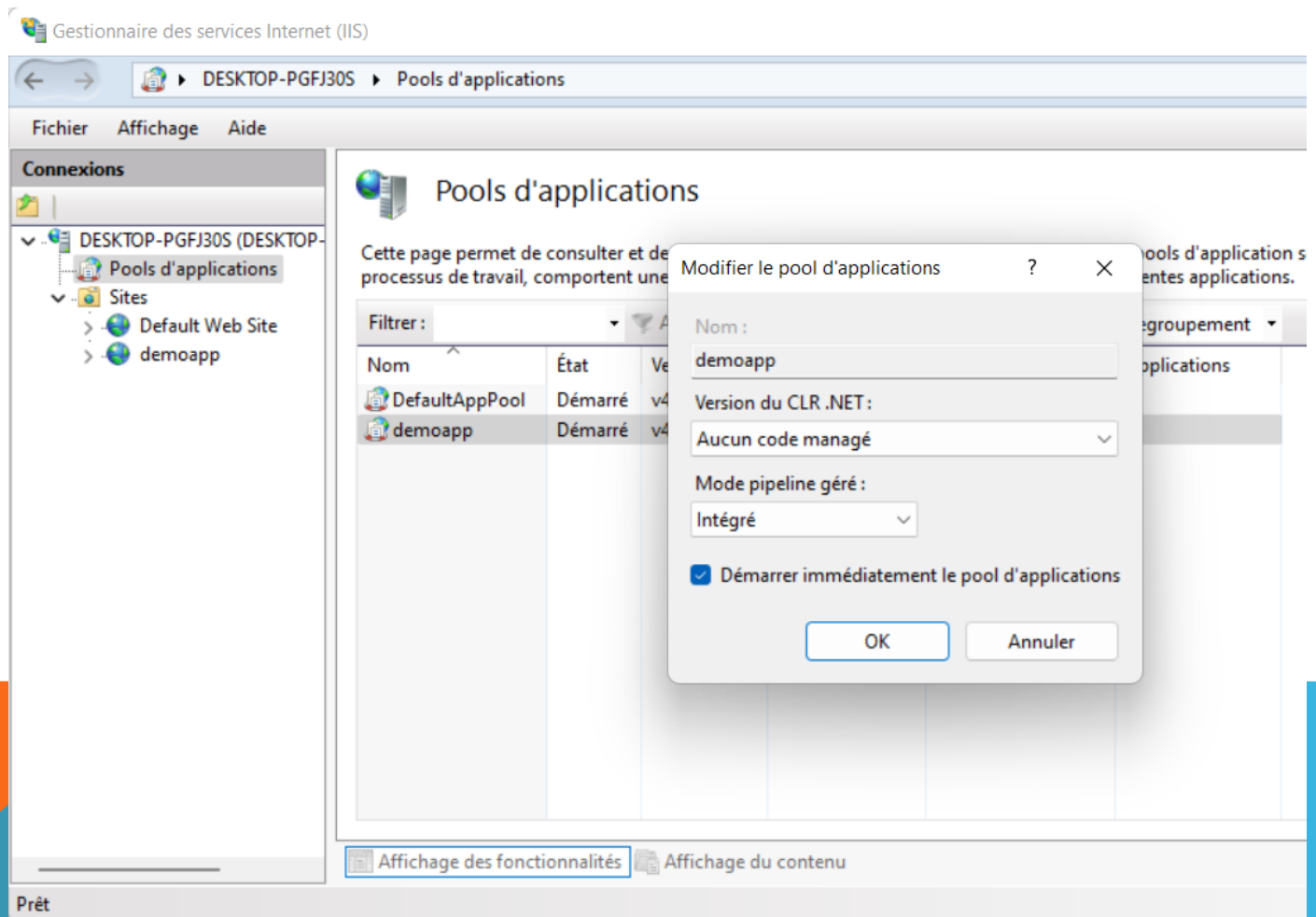
- Création du site web sous IIS

The screenshot shows the 'Ajouter un site Web' (Add Website) dialog box in IIS Express. The dialog is titled 'Ajouter un site Web' and has a close button (X) and a help button (?). It contains the following fields and buttons:


- Nom du site :** A text box containing 'demoapp'.
- Pool d'applications :** A text box containing 'demoapp' and a 'Sélectionner...' button.
- Répertoire de contenu** (Content Directory):
  - Chemin d'accès physique :** A text box containing 'C:\dev\DemoApp' and a browse button (...).
  - Authentification directe** (Direct Authentication):
    - A button labeled 'Se connecter en tant que...' (Connect as...).
    - A button labeled 'Tester les paramètres...' (Test settings...).
- Liaison** (Binding):
  - Type :** A dropdown menu showing 'http'.
  - Adresse IP :** A dropdown menu showing 'Toutes non attribuées' (All unassigned).
  - Port :** A text box containing '80'.
  - Nom de l'hôte :** A text box containing 'demoapp.fr'.
  - Exemple :** A text box containing 'www.contoso.com ou marketing.contoso.com'.

# HÉBERGEMENT AVEC IIS EXPRESS

- Création du site web sous IIS



## HÉBERGEMENT UNE PAGE GITHUB

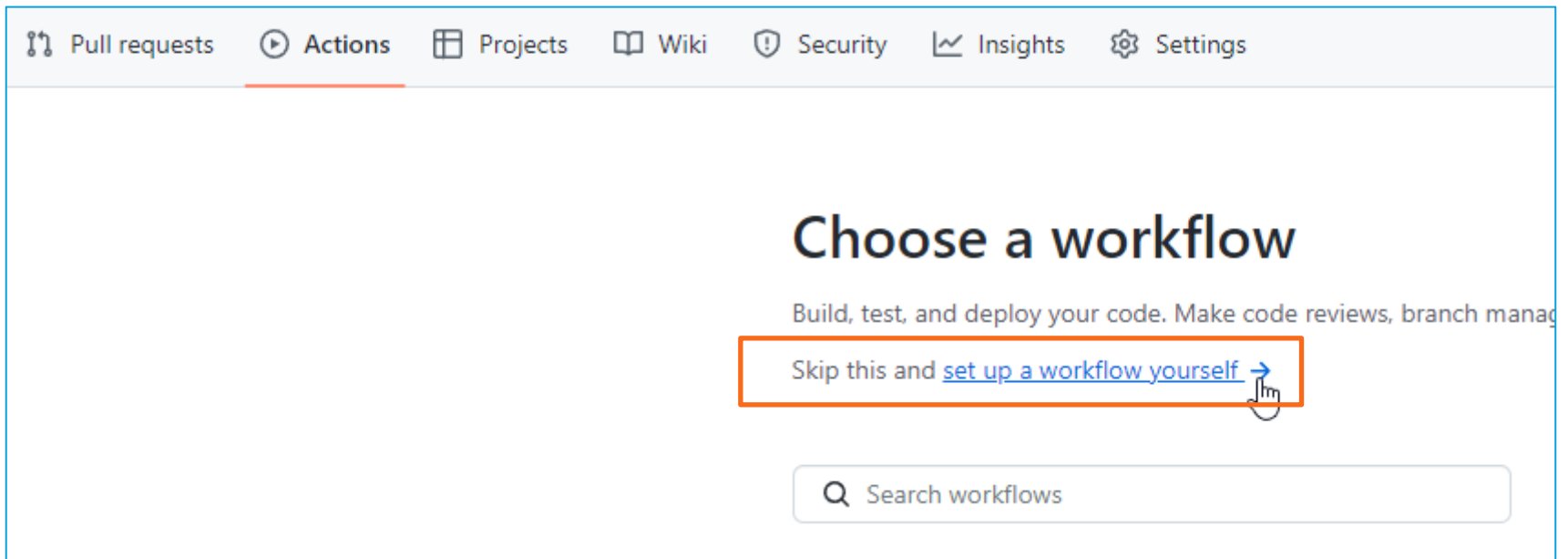
-  Uniquement contenu statique
  - Impossible d'héberger une application Blazor Server, uniquement WASM
- Possibilité de faire des actions sur une branche pour :
  - Faire de la validation de code
  - Déployer sur un hébergement externe (Azure, Kubernetes, etc...)
  - Déployer sur une page github
  - Faire du déploiement continu, etc...



## **HÉBERGEMENT UNE PAGE GITHUB**

- Mise en place :
- Pusher le code sur un repository Github
- Ajouter une action de publish automatique sur une branche
- Mettre le code publié sur une branche spécifique
- Définir cette branche comme source pour la page Github

# HÉBERGEMENT UNE PAGE GITHUB



# HÉBERGEMENT UNE PAGE GITHUB

```
1  name: DemoApp
2
3  # Run workflow on every push to the master branch
4  on:
5    push:
6      branches: [ master ]
7
8  jobs:
9    deploy-to-github-pages:
10     # use ubuntu-latest image to run steps on
11     runs-on: ubuntu-latest
12     steps:
13       # Publish project
14       - uses: actions/checkout@v3
15       - name: Setup .NET Core SDK
16         uses: actions/setup-dotnet@v2
17         with:
18           dotnet-version: 6.0.x
19       - run: dotnet publish -c Release -o release --nologo
20
21       # changes the base-tag in index.html from '/' to 'DemoApp' to match GitHub Pages repository subdirectory
22       - name: Change base-tag in index.html from / to DemoApp
23         run: sed -i 's/<base href="\/" \/>/<base href="\DemoApp\/" \/>/g' release/wwwroot/index.html
24
25       # copy index.html to 404.html to serve the same file when a file is not found
26       - name: copy index.html to 404.html
27         run: cp release/wwwroot/index.html release/wwwroot/404.html
28
29       # add .nojekyll file to tell GitHub pages to not treat this as a Jekyll project. (Allow files and folders starting with an underscore)
30       - name: Add .nojekyll file
31         run: touch release/wwwroot/.nojekyll
32
33       - name: Commit wwwroot to GitHub Pages
34         uses: JamesIves/github-pages-deploy-action@3.7.1
35         with:
36           GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }
37           BRANCH: gh-pages
38           FOLDER: release/wwwroot
```



# HÉBERGEMENT UNE PAGE GITHUB

Triggered via push 3 hours ago

Status

Total duration

Artifacts

 RmyK pushed  a03c733 master


Success

1m 16s

—

main.yml

on: push

 deploy-to-github-pages

1m 7s

# HÉBERGEMENT UNE PAGE GITHUB

[Security](#) [Insights](#) [Settings](#)

⚙️ General

Access

👤 Collaborators

💬 Moderation options

Code and automation

🔑 Branches

🏷️ Tags

🔄 Actions

🔗 Webhooks

📁 Environments

**📄 Pages**

## GitHub Pages

GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.

✓ Your site is published at <https://rmyk.github.io/DemoApp/>

### Source

Your GitHub Pages site is currently being built from the `gh-pages` branch. [Learn more.](#)

🔑 Branch: `gh-pages`

📁 `/ (root)`

Save

### Custom domain

Custom domains allow you to serve your site from a domain other than `rmyk.github.io`. [Learn more.](#)

Save

Remove

# HÉBERGEMENT UNE PAGE GITHUB

2 workflow runs

Event ▾

Status ▾

Branch ▾

Actor ▾



**pages build and deployment**

pages-build-deployment #1: by github-pages bot



3 hours ago



33s

...



**Update main.yml**

DemoApp #2: Commit a03c733 pushed by RmyK

master



3 hours ago



1m 16s

...

QUESTIONS ? REMARQUES ?



# DÉMONSTRATION





# MODULE#12

LES CONTRÔLEUR API



## LES CONTRÔLEURS API (WEB API)

- Web Api utilise le routage par verbe HTTP :
  - GET
  - POST
  - PUT
  - DELETE
- Va utiliser un jeu d'attribut pour paramétrer le routage
  - Au niveau du contrôleur

```
[ApiController]  
[Route("[controller]")]  
public class WeatherForecastController : ControllerBase
```

- Au niveau des méthodes d'action

```
[HttpGet]  
[Route("customers/{customerId}/orders")]  
public IEnumerable<Order> FindOrdersByCustomer(int customerId)
```

## LE FORMATAGE

- Séri­a­li­sa­tion des don­nées en JSON ou XML
- Appelant choisi le format via l'entête HTTP Accept (Json ou Xml)
- Pilotage des données sérialisés par attribut
  - JSON uniquement :
  - Pour ignorer des données :

```
[JsonIgnore]
```

```
public int ProductCode { get; set; } // omitted
```

- Pour JSON et XML:
  - Tout est automatique ignoré sauf les propriétés décorées avec [DataMember]

```
[DataContract]
```

```
public class Product  
{
```

```
    [DataMember]
```

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

# GESTION DES DONNÉES

- Utilisation de Try/Catch
- Utilisation des objets du framework
  - Ok() : pour renvoyer automatique un HTTP 200
  - Problem() : une erreur personnalisée
    - Possibilité de choisir le code en fonction du type d'erreur à remonter
    - Message d'erreur personnalisé
  - NotFound() : pour indiquer une ressource non trouvé
  - Forbid() : requête interdite
  - BadRequest(): mauvaise requête
  - Etc ...

QUESTIONS ? REMARQUES ?



# DÉMONSTRATION



# MODULE#13

EFCORE



## ENTITY FRAMEWORK

- ORM : Object-relational mapping
  - Fait le lien entre le monde relationnel (table, colonne, ligne => BDD) et le monde objet (classe, héritage, etc)
- Open Source
- Pour l'environnement .NET, soutenu et développé par Microsoft
- Historiquement 2 modes de liaison à la base de données
  - Database First
    - N'existe plus en version Core
  - Code First
    - Permet les migrations

```
C:\Users\R my>dotnet ef  
  
-----  
Entity Framework Core .NET Command-line Tools 6.0.3
```



## **EF CORE VS EF 6**

- EF Core :
  - Créer pour .NET Core et parfaitement intégré au outil .NET Core (comme ASP.NET Core)
  - Jusqu'à la version 3.1 compatible avec .NET Core et .NET Framework via .NET Standard
  - Développeur actif de la par de la communauté de Microsoft
  - Plus de fonctionnalité de EF6 même si certaines manques
- EF6:
  - Créer pour .NET Framework
  - Dernière version fonctionne sur .NET Core et .NET Framework via le multi-ciblage
  - Développement arrêté par Microsoft

<https://docs.microsoft.com/fr-fr/ef/efcore-and-ef6/>

## CONTEXT EF CORE

- Définition d'une classe héritant de DbContext
- Classe qui va permettre de manipuler les données
- Va contenir les DbSet<T> contenant les objets de la base
- Peut se configurer directement dans la classe

```
public class ApplicationDbContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Test");
    }
}
```

- Via le mécanisme d'injection de dépendance

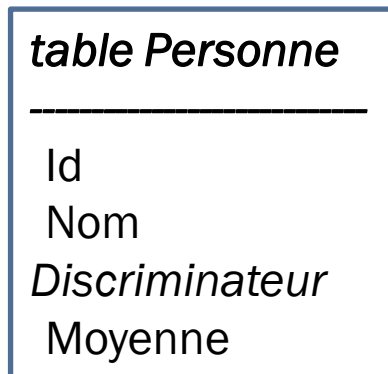
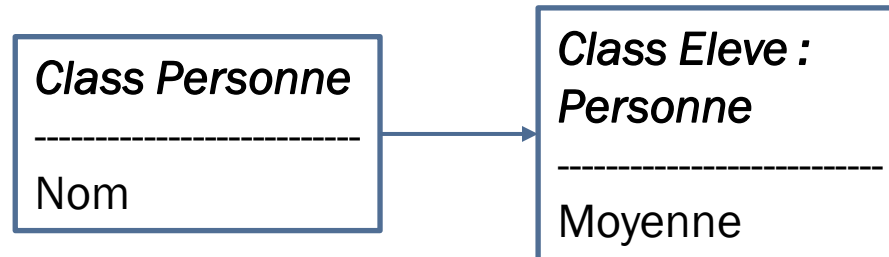
```
services.AddDbContext<ApplicationDbContext>(
    options => options.UseSqlServer("name=ConnectionStrings:DefaultConnection"));
```

## FUNCTIONNEMENT ENTITY FRAMEWORK

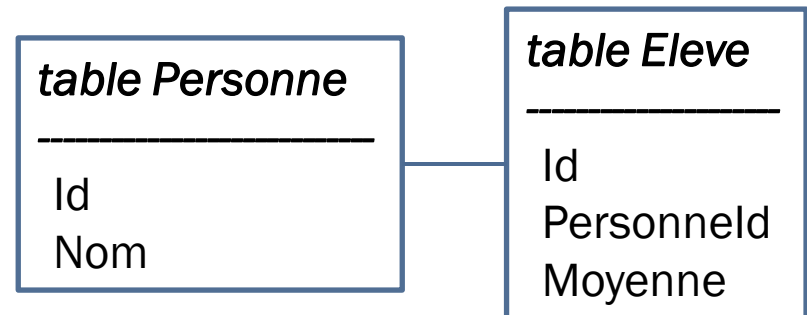
- Mécanisme d'attribut
- OnModelCreating
- ModelSeed
- Fonctionne de manière transactionnelle (SaveChanges())
- Gestion des accès concurrentiels via attributs

```
[ConcurrencyCheck]  
Oreferences  
public DateTime UpdateDate { get; set; }
```

## GESTION DE L'HÉRITAGE DANS EF



TPH : Table per Hierarchy



TPT : Table per Type

# CRUD EFCORE

- **CRUD**
  - **Create** : Ajoute des données en base (Insert)
  - **Read** : Lit des données en base (Select)
  - **Update** : Met à jour des données dans la base (Update)
  - **Delete** : Supprimer des données en base (Delete)
  - = 4 action de base de manipulation de données

```
public static void Insert(Personne pers)
{
    using (ProjEcoleDbContext ctx = new ProjEcoleDbContext())
    {
        pers.DateUpdate = DateTime.Now;
        ctx.Personnes.Add(pers);
        ctx.SaveChanges();
    }
}
```

## BONNE PRATIQUE EFCORE

- Utilisation d'un repository
- Une instance **DbContext** unique par fonction
  - A sortir de l'injection de dépendance
  - Valable pour une majorité de cas
  - Attention au mécanisme de tracking d'EF

```
public class Repository
{
    1reference
    public void Insert<T>(T entity) where T : class
    {
        using var _dbctx = new EcoleDbContext();
        _dbctx.Set<T>().Add(entity);
        _dbctx.SaveChanges();
    }

    1reference
    public List<T> GetAll<T>() where T : class
    {
        using var _dbctx = new EcoleDbContext();
        return _dbctx.Set<T>().ToList();
    }

    1reference
    public void Delete<T>(T entity) ...

    1reference
    public T GetById<T>(int id) ...
}
```

QUESTIONS? REMARQUES?



# DÉMONSTRATION

