Blazor WebAssembly

m2iformation.fr

Comprendre WebAssembly - Introduction à WebAssembly

- WebAssembly (WASM) est un format de code binaire pour un pile d'exécution sécurisée et portable, visant à permettre aux programmes écrits dans de multiples langages de s'exécuter avec des performances proches de celles du code natif sur le web. Sa conception repose sur plusieurs principes clés, et son histoire reflète une évolution significative dans le développement web.
- **Origine** : L'idée de WebAssembly a émergé dans les années 2010, résultant de la reconnaissance des limitations de JavaScript en termes de performance pour certaines applications web complexes, telles que les jeux 3D, la manipulation de médias et les simulations scientifiques.
- **Premiers pas** : Des projets tels qu'asm.js ont pavé la voie en démontrant qu'il était possible d'atteindre des performances proches du code natif dans les navigateurs en optimisant JavaScript pour des cas d'utilisation spécifiques. Cependant, asm.js présentait des limitations, notamment une taille de code importante et un temps de parsing non négligeable.
- **Naissance officielle** : WebAssembly est officiellement né avec le support et la collaboration des principaux éditeurs de navigateurs (Mozilla, Google, Microsoft, et Apple), soulignant un rare consensus dans l'industrie du développement web. La première version (MVP) de WebAssembly a été lancée en 2017.

Comprendre WebAssembly - Principes de base de WebAssembly

Performance et sécurité :

- **Conçu pour la vitesse** : WebAssembly vise à exécuter du code à une vitesse proche de celle du code natif en exploitant les capacités matérielles de la machine hôte.
- **Sandboxing** : WebAssembly est exécuté dans un environnement isolé (sandbox), ce qui limite les risques de sécurité et protège les données de l'utilisateur.

Interopérabilité avec JavaScript :

- Bien que WASM soit conçu pour surpasser JavaScript en termes de performances, il est également conçu pour travailler en tandem avec lui, permettant une interaction fluide entre le code WASM et JavaScript. Cela permet aux développeurs de tirer parti des meilleures caractéristiques de chaque technologie.

Portabilité et indépendance du langage :

- **Multiplateforme** : WebAssembly est conçu pour être exécuté sur n'importe quel système d'exploitation ou appareil disposant d'un navigateur web moderne, sans modification.
- **Support multilangue** : WASM permet aux développeurs d'utiliser des langages autres que JavaScript, comme C, C++, Rust et même Python, pour le développement web. Les langages compilés en WASM peuvent s'exécuter dans le navigateur, élargissant les possibilités pour le développement d'applications web.

Comprendre WebAssembly - Cas d'usage et avantages par rapport à JavaScript

Applications de Jeux :

- WASM permet le développement de jeux vidéo pour le web avec des performances proches de celles des jeux natifs, grâce à sa capacité à exécuter du code à haute vitesse et à utiliser efficacement les ressources système.

Applications de Traitement d'Images et de Vidéos :

- Le traitement d'images et de vidéos en temps réel, y compris le rendu 3D et les effets spéciaux, bénéficie grandement de WebAssembly. WASM peut gérer des calculs lourds nécessaires pour ces tâches bien mieux que JavaScript.

Simulation et Modélisation Scientifique :

- Les simulations scientifiques et les modélisations, qui nécessitent de lourds calculs mathématiques, peuvent s'exécuter plus efficacement avec WebAssembly.

Portage d'Applications Existantes sur le Web :

- WebAssembly facilite le portage d'applications existantes écrites dans des langages tels que C/C++ vers le web, sans réécriture complète en JavaScript.

Comprendre WebAssembly - Cas d'usage et avantages par rapport à JavaScript

Performances Accrues :

- **Exécution plus rapide** : WASM permet une exécution proche de la vitesse du code natif, surpassant JavaScript dans les tâches calculatoires intensives.
- **Chargement plus rapide** : Le format binaire de WASM se charge plus rapidement que le code JavaScript équivalent, améliorant ainsi le temps de démarrage des applications.

Sécurité Renforcée :

- **Environnement sandbox** : Tout comme JavaScript, WASM s'exécute dans un environnement sandbox dans le navigateur, limitant l'accès au système d'exploitation sous-jacent et offrant une couche de sécurité supplémentaire.

Compatibilité et Portabilité :

- **Large support navigateur** : WASM est supporté par tous les navigateurs modernes, garantissant une expérience utilisateur cohérente à travers différentes plateformes.
- **Indépendance du langage** : Les développeurs peuvent utiliser des langages autres que JavaScript, adaptés à leurs besoins spécifiques ou à leur expertise, pour développer des applications web.

Intégration et utilisation de WebAssembly avec Blazor.

L'intégration et l'utilisation de WebAssembly (WASM) avec Blazor représentent une avancée significative dans le développement d'applications web modernes, permettant aux développeurs .NET de construire des applications riches côté client en utilisant C# et d'autres langages .NET, sans recourir à JavaScript.
Blazor est un framework de Microsoft pour construire des interfaces utilisateur web interactives avec C#. Il offre deux modèles de hosting principaux : Blazor Server et Blazor WebAssembly.

- **Blazor Server** : Exécute le code côté serveur et communique avec le client via SignalR.
- **Blazor WebAssembly** : Exécute le code C# directement dans le navigateur grâce à WebAssembly.

L'approche WebAssembly est particulièrement révolutionnaire car elle permet d'exécuter du code .NET dans le navigateur, à une vitesse proche de celle du code natif, sans plugins supplémentaires.

- Lorsque vous créez une application Blazor WebAssembly, le compilateur .NET convertit le code C# en un format intermédiaire (IL), qui est ensuite compilé en WebAssembly. Ce processus permet au code .NET de s'exécuter dans le navigateur.
- Le runtime .NET, les bibliothèques requises, et l'application Blazor elle-même sont téléchargés dans le navigateur lors de la première visite de l'utilisateur. Le code s'exécute ensuite entièrement côté client, offrant une expérience utilisateur rapide et réactive.

Intégration et utilisation de WebAssembly avec Blazor.

Avantages de l'intégration

- **Performance** : Grâce à WASM, les applications Blazor WebAssembly peuvent offrir des performances proches de celles des applications natives.
- **Développement unifié** : Les développeurs peuvent utiliser C# et .NET pour le développement front-end et back-end, simplifiant le processus de développement et réduisant le besoin de context-switching.
- **Richesse de l'écosystème .NET** : Accès à une vaste gamme de bibliothèques et outils .NET pour le développement d'applications.
- **Interopérabilité JavaScript** : Bien que Blazor WebAssembly permette de minimiser l'usage de JavaScript, il offre également la possibilité d'interagir avec des bibliothèques et API JavaScript, offrant le meilleur des deux mondes.

Syntaxe de base Blazor.

Les composants Blazor sont au cœur du développement d'applications avec le framework Blazor, permettant une construction modulaire et réactive des interfaces utilisateur web en utilisant C# et Razor. Un composant Blazor peut être compris comme une combinaison de balisage HTML et de logique C# encapsulée, qui peut être réutilisée dans différentes parties de l'application.

1. **Création d'un Composant Blazor**

Un composant Blazor est généralement défini dans un fichier avec une extension `.razor`. Le composant peut contenir du HTML statique, des directives Razor pour la dynamique, et du code C# pour la logique.

Exemple simple d'un composant Blazor (HelloWorld.razor) :

```
@page "/hello"

<h1>Hello, World!</h1>

@code {
    // C# Code pour la logique du composant ici
}
```

Syntaxe de base Blazor.

2. Utilisation des Paramètres de Composant

Les composants peuvent accepter des paramètres pour personnaliser leur contenu ou leur comportement. Les paramètres de composant sont définis en utilisant des propriétés publiques annotées avec l'attribut `[Parameter]`.

```
<p>Hello, @Name!</p>

@code {
    [Parameter]
    public string Name { get; set; }
}
```

Vous pouvez utiliser ce composant dans un autre composant en lui passant un nom :

```
<Greeting Name="World" />
```

Syntaxe de base Blazor.

3. **Data Binding**

Le data binding permet d'établir une connexion interactive entre les propriétés du composant et les éléments de l'interface utilisateur.

Exemple de data binding :

```
<input @bind="name" />

<p>Hello, @name!</p>

@code {
    private string name = "World";
}
```

Syntaxe de base Blazor.

Gestion des Événements

Les composants Blazor peuvent réagir aux événements du DOM en utilisant la syntaxe `@onEVENT`, où `EVENT` est le nom de l'événement.

Exemple de gestion d'un clic de bouton :

```
<button @onclick="OnClick">Click me</button>

<p>@message</p>

@code {
    private string message;

    private void OnClick()
    {
        message = "Button clicked!";
    }
}
```

Exercice 1

Créez un composant Blazor WebAssembly nommée "Le Juste Prix". Dans ce jeu, l'utilisateur doit deviner le prix correct d'un objet mystère généré aléatoirement par l'application. Le jeu guide l'utilisateur vers le prix correct en lui indiquant après chaque tentative si le prix mystère est plus élevé ou plus bas que sa devinette. Le jeu se termine lorsque l'utilisateur trouve le juste prix.

Routing avec Blazor

1. Configuration du Routage

Dans une application Blazor, le routage est configuré en utilisant le composant `Router` qui est généralement défini dans le fichier `App.razor`. Ce composant `Router` écoute les changements d'URL et charge le composant correspondant à l'URL actuelle.

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

Le paramètre `AppAssembly` indique au `Router` où chercher les composants marqués avec l'attribut `@page`, qui sont les points d'entrée du routage.

2. Définition des Routes

Pour définir une route dans un composant Blazor, vous utilisez l'attribut `@page` suivi de la route. Par exemple, pour créer une page qui répond à l'URL `/hello`, vous ajouteriez `@page "/hello"` au début du fichier composant.

```
@page "/hello"
<h1>Hello, World!</h1>
```

```
@page "/user/{UserId}"
<h1>User @UserId</h1>
@code {
```

Routing avec Blazor

3. Navigation

Pour naviguer entre les pages, Blazor fournit le composant `NavLink` qui génère un lien HTML (`<a href="...">`). Le composant `NavLink` ajoute une classe CSS active lorsque l'URL correspond à la destination du lien, ce qui est utile pour mettre en évidence le lien actif dans une barre de navigation.

```
<NavLink href="/hello">Say Hello</NavLink>
<NavLink href="/about">About Us</NavLink>
```

Pour la navigation programmatique, vous pouvez injecter le service `NavigationManager` et utiliser sa méthode `NavigateTo` :

```
@inject NavigationManager NavigationManager

<button @onclick="NavigateToHello">Say Hello</button>

@code {
    void NavigateToHello()
    {
        NavigationManager.NavigateTo("/hello");
    }
}
```

Exercice 2

Créez une application de "Galerie Photo" interactive en Blazor WebAssembly, où les utilisateurs peuvent naviguer entre différentes catégories de photos (par exemple, Nature, Villes, Espace). Utilisez le routage de Blazor pour changer de catégorie sans recharger la page.

Cycle de vie

Dans Blazor, chaque composant suit un cycle de vie spécifique, qui est une série d'étapes par lesquelles le composant passe de sa création à sa destruction. Comprendre ce cycle de vie est crucial pour gérer efficacement les ressources, exécuter la logique au bon moment, et optimiser les performances de l'application.

1. **OnInitAsync et OnInit**

- `OnInitAsync` : Cette méthode asynchrone est appelée après l'initialisation du composant mais avant son rendu. Elle est idéale pour effectuer des opérations asynchrones, comme la récupération de données à partir d'une API. Si `OnInitAsync` est surchargée, `OnInit` n'est généralement pas utilisée.
- `OnInit` : Similaire à `OnInitAsync` mais pour les opérations synchrones. Utilisée pour initialiser les données du composant qui ne nécessitent pas d'opérations asynchrones.

2. **OnParametersSetAsync et OnParametersSet**

- `OnParametersSetAsync` : Appelée après que Blazor a attribué des paramètres au composant et après `OnInitAsync`. Utilisez cette méthode pour une initialisation basée sur les paramètres, en particulier pour les opérations asynchrones qui dépendent des valeurs de paramètre.
- `OnParametersSet` : La version synchrone de `OnParametersSetAsync`, appelée immédiatement après pour les mises à jour synchrones basées sur les paramètres.

Cycle de vie

3. **OnAfterRenderAsync et OnAfterRender**

- `OnAfterRenderAsync` : Exécutée après le rendu du composant. Cette méthode est particulièrement utile pour exécuter du code JavaScript via l'interopérabilité JavaScript ou pour effectuer des actions qui nécessitent que le DOM soit complètement rendu. Elle dispose d'un paramètre `firstRender` qui indique si c'est le premier rendu du composant.
- `OnAfterRender` : La version synchrone de `OnAfterRenderAsync`. Comme son homologue asynchrone, elle est appelée après le rendu du composant.

4. **ShouldRender**

- `ShouldRender` : Méthode appelée pour déterminer si le composant doit être re-rendu. En surchargeant cette méthode, vous pouvez contrôler le processus de rendu en fonction de la logique personnalisée, ce qui peut aider à améliorer les performances de l'application.

Cycle de vie

```
@implements IDisposable
@code {
    protected override async Task OnInitAsync()
    {
        // Appelé à l'initialisation du composant pour charger des données.
    }
    protected override bool ShouldRender()
    {
        // Détermine si le composant doit être re-rendu.
        return true; // Retourne false pour empêcher le rendu.
    }

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            // Code exécuté une seule fois après le premier rendu du composant.
        }
    }
    public void Dispose()
    {
        // Nettoyage des ressources.
```

Exercice

Créez une application Blazor WebAssembly intitulée "Explorateur de Produits Alimentaires" qui permet aux utilisateurs de rechercher des informations sur des produits alimentaires en utilisant l'API d'OpenFoodFacts. Cette application devrait permettre aux utilisateurs d'entrer le nom ou le code-barres d'un produit alimentaire et afficher les détails de ce produit, tels que les ingrédients, les valeurs nutritionnelles, et les labels de qualité.

MVC et MVVM en Blazor

Les patterns MVC (Modèle-Vue-Contrôleur) et MVVM (Modèle-Vue-VueModèle) sont deux architectures logicielles couramment utilisées dans le développement d'applications, y compris les applications web modernes. L'intégration de ces modèles dans des projets Blazor, en particulier avec Blazor WebAssembly, nécessite une compréhension de la manière dont ces patterns peuvent s'adapter à l'environnement de programmation de Blazor.

1. **MVC avec Blazor**

Le pattern MVC sépare l'application en trois composants principaux : le modèle (les données), la vue (l'interface utilisateur), et le contrôleur (la logique métier qui fait le lien entre le modèle et la vue). Bien que Blazor ne suive pas strictement l'architecture MVC, il est possible de l'adapter en suivant ces principes :

- **Modèle** : Définit les structures de données de l'application, souvent en tant que classes C#. Ces modèles peuvent représenter des entités de base de données, des structures de données en mémoire, etc.
- **Vue** : Dans Blazor, les composants Razor (.razor) agissent comme des vues, définissant l'interface utilisateur et la logique d'affichage. Les composants Razor peuvent intégrer à la fois la marque HTML et le code C# pour créer des interfaces dynamiques.
- **Contrôleur** : Blazor ne possède pas de contrôleur dans le sens traditionnel du MVC. Toutefois, la logique qui serait normalement située dans les contrôleurs peut être implémentée à travers des services C# qui gèrent les interactions entre les modèles et les vues (composants Razor). Ces services peuvent gérer la logique métier, l'accès aux données, etc.

MVC et MVVM en Blazor

2. **MVVM avec Blazor**

Le pattern MVVM est particulièrement bien adapté à Blazor, surtout pour des applications complexes. Il propose une séparation des préoccupations similaire au MVC, mais avec une couche supplémentaire qui facilite la liaison de données (data binding) entre la vue et le modèle :

- **Modèle** : Comme dans MVC, représente les structures de données.
- **Vue** : Les composants Razor forment la vue, affichant les données à l'utilisateur et capturant les interactions utilisateur.
- **VueModèle (ViewModel)** : Une couche intermédiaire entre la vue et le modèle. Le ViewModel expose des propriétés et des commandes que la vue peut lier, permettant une interaction dynamique avec les données sans nécessiter que la vue connaisse la logique métier sous-jacente. Le ViewModel réagit aux commandes de la vue (par exemple, un clic de bouton) et manipule les modèles en conséquence.

Organisation du projet

Pour bien respecter le pattern **MVVM**, il est recommandé de structurer le projet de cette manière :

```
/Pages
  - MainPage.razor (View)
  - DetailsPage.razor (View)
/ViewModels
  - MainPageViewModel.cs (ViewModel)
  - DetailsPageViewModel.cs (ViewModel)
/Models
  - Product.cs (Model)
```

Cette organisation permet de séparer chaque couche :

- **Models** : Représentation des entités de données (objets métiers, DTO, etc.).
- **ViewModels** : Gestion de la logique de l'interface utilisateur (champs, propriétés, actions, validation, etc.).
- **Views** : Pages et composants Blazor (.razor) qui affichent les informations et permettent les interactions.

Mise en œuvre du MVVM en Blazor

Étape 1 : Le modèle (Model)

Le **Model** est la représentation des **données**. Il peut s'agir d'une simple classe C#.

```
// Models/Product.cs
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

Étape 2 : La ViewModel (ViewModel)

Le **ViewModel** est une classe C# qui contient :

- **Les propriétés** que la vue (View) peut afficher.
- **Les méthodes** qui contiennent la logique de l'interface utilisateur.
- **L'implémentation de INotifyPropertyChanged** (optionnel) pour **notifier les modifications** de propriétés.

> ⚠️ Blazor étant basé sur le **data binding unidirectionnel**, il n'a pas de "Two-Way Binding" natif comme dans WPF. Mais on peut le simuler avec `EventCallback` et `@bind`.

```
// ViewModels/MainPageViewModel.cs
public class MainPageViewModel
{
    // Liste des produits
    public List<Product> Products { get; set; }

    // Propriété pour la saisie d'un nouveau produit
    public Product NewProduct { get; set; } = new Product();

    public MainPageViewModel()
    {
        // Initialisation de quelques données
        Products = new List<Product>
        {
            new Product { Id = 1, Name = "Produit A", Price = 20.50m },
            new Product { Id = 2, Name = "Produit B", Price = 10.00m },
            new Product { Id = 3, Name = "Produit C", Price = 5.75m }
        };
    }

    // Méthode pour ajouter un produit
    public void AddProduct()
    {
        if (!string.IsNullOrWhiteSpace(NewProduct.Name) && NewProduct.Price > 0)
        {
            NewProduct.Id = Products.Count + 1;
            Products.Add(new Product
            {
                Id = NewProduct.Id,
                Name = NewProduct.Name,
                Price = NewProduct.Price
            });

            // On réinitialise le produit saisi
            NewProduct = new Product();
        }
    }
}
```

Étape 3 : La Vue (View)

La **View** est le composant Blazor (`.razor`) qui interagit avec l'utilisateur. On va injecter le **ViewModel** et l'utiliser pour accéder aux propriétés et méthodes.

```
<!-- Pages/MainPage.razor -->
@page "/mainpage"
@inject MainPageViewModel ViewModel

<h1>Liste des produits</h1>

<table class="table">
    <thead>
        <tr>
            <th>Id</th>
            <th>Nom</th>
            <th>Prix</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var product in ViewModel.Products)
        {
            <tr>
                <td>@product.Id</td>
                <td>@product.Name</td>
                <td>@product.Price €</td>
            </tr>
        }
    </tbody>
</table>

<h2>Ajouter un produit</h2>

<div>
    <label>Nom :</label>
    <input type="text" @bind="ViewModel.NewProduct.Name" />

    <label>Prix :</label>
```

3. Enregistrement du ViewModel dans le conteneur DI

Pour que Blazor puisse injecter le **ViewModel** dans la **View**, il faut l'enregistrer dans le **conteneur d'injection de dépendance** (DI).

```
// Program.cs
builder.Services.AddSingleton<MainPageViewModel>();
```

Fonctionnalités avancées

1 Notification des changements (INotifyPropertyChanged)

Si vous voulez que la Vue se mette à jour **dès qu'une propriété change**, vous devez implémenter l'interface `INotifyPropertyChanged` dans le **ViewModel**.

```
public class MainPageViewModel : INotifyPropertyChanged
{
    private Product _newProduct;
    public event PropertyChangedEventHandler PropertyChanged;
    public List<Product> Products { get; set; } = new List<Product>();

    public Product NewProduct
    {
        get => _newProduct;
        set
        {
            _newProduct = value;
            OnPropertyChanged(nameof(NewProduct));
        }
    }

    public MainPageViewModel()
    {
        Products.Add(new Product { Id = 1, Name = "Produit A", Price = 20.50m });
    }

    public void AddProduct()
    {
        Products.Add(new Product { Id = Products.Count + 1, Name = NewProduct.Name, Price = NewProduct.Price });
        NewProduct = new Product();
        OnPropertyChanged(nameof(Products));
    }

    protected void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

2 Commande et validation des formulaires

On peut valider les champs et afficher des erreurs. Blazor permet de **valider des formulaires** via `EditForm` et `DataAnnotations`.

```
public class Product
{
    public int Id { get; set; }

    [Required(ErrorMessage = "Le nom est requis")]
    [StringLength(50, ErrorMessage = "Le nom ne doit pas dépasser 50 caractères")]
    public string Name { get; set; }

    [Range(0.01, 1000, ErrorMessage = "Le prix doit être compris entre 0,01 et 1000")]
    public decimal Price { get; set; }
}
```

Dans la vue, on utilise **EditForm** et **ValidationSummary**.

```
<EditForm Model="ViewModel.NewProduct" OnValidSubmit="ViewModel.AddProduct">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <div>
        <label>Nom :</label>
        <input type="text" @bind="ViewModel.NewProduct.Name" />
    </div>

    <div>
        <label>Prix :</label>
        <input type="number" @bind="ViewModel.NewProduct.Price" step="0.01" />
    </div>

    <button type="submit">Ajouter</button>
</EditForm>
```

MVVM

Composant	Rôle
Model	Représente les données de l'application (Product).
ViewModel	Contient la logique de présentation, l'état et les actions.
View	Interface utilisateur (HTML/CSS) qui affiche les données et interagit avec ViewModel.

Les points importants

- La **vue (View)** ne contient pas de logique, elle appelle simplement les actions du **ViewModel**.
- Le **ViewModel** contient l'état et la logique de l'interface utilisateur.
- Les données sont représentées par les **Models**.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Communication composants

Communication Parent-Enfant (Data Binding)

Cette méthode est la plus courante. Le composant parent transmet des **données** au composant enfant via des **paramètres de composant**.

1. **Dans le composant enfant** :
   - On utilise l'attribut `[Parameter]` pour indiquer qu'une propriété du composant peut recevoir une valeur depuis le parent.

```
@code {
    [Parameter]
    public string Message { get; set; }
}
```

   - Dans le composant enfant (fichier `.razor`), on affiche la valeur reçue :

```
<p>Message du parent : @Message</p>
```

2. **Dans le composant parent** :
   - On utilise le composant enfant dans le parent en lui passant une valeur :

```
<ChildComponent Message="Hello from Parent" />
```

Exemple complet

Composant Parent

```
<ChildComponent Message="Bonjour, je viens du parent !" />
```

Composant Enfant (ChildComponent.razor)

```
<p>Message reçu : @Message</p>

@code {
    [Parameter]
    public string Message { get; set; }
}
```

2. Communication Parent-Enfant (EventCallback)

Si le composant enfant doit **notifier le parent** lorsqu'un événement se produit (par exemple, lorsqu'un bouton est cliqué), on utilise `EventCallback`.

1. **Dans le composant enfant** :
   - On définit un `EventCallback` dans le composant enfant.

```
[Parameter]
public EventCallback OnButtonClick { get; set; }
```

   - Lors d'une action (comme un clic sur un bouton), on déclenche cet événement :

```
<button @onclick="OnClick">Cliquer</button>

@code {
    private async Task OnClick()
    {
        await OnButtonClick.InvokeAsync();
    }
}
```

2. **Dans le composant parent** :
    - On passe la méthode à exécuter lorsque l'événement est déclenché :

```
<ChildComponent OnButtonClick="HandleButtonClick" />

@code {
    private void HandleButtonClick()
    {
        Console.WriteLine("Bouton cliqué dans le composant enfant !");
    }
}
```

3. Communication Frère-Frère (via un composant parent)

Les **composants frères** ne peuvent pas se communiquer directement, mais ils peuvent le faire **via leur parent commun**.

1. Les composants enfants envoient des **informations au parent** via `EventCallback`.
2. Le parent transmet ces **informations aux autres enfants** via des paramètres.

Exemple complet

Composant Parent

```
<ChildA OnDataChange="HandleDataChange" />
<ChildB DataFromA="@DataFromA" />

@code {
    private string DataFromA;

    private void HandleDataChange(string data)
    {
        DataFromA = data;
    }
}
```

Exemple complet

Composant Enfant A (ChildA.razor)

```
<button @onclick="SendData">Envoyer des données</button>

@code {
    [Parameter]
    public EventCallback<string> OnDataChange { get; set; }

    private async Task SendData()
    {
        await OnDataChange.InvokeAsync("Données depuis ChildA");
    }
}
```

Exemple complet

Composant Enfant B (ChildB.razor)

```
<p>Data reçue : @DataFromA</p>

@code {
    [Parameter]
    public string DataFromA { get; set; }
}
```

4. Communication Globale (Service Singleton / DI)

Si vous souhaitez **partager des données globales** entre des composants qui ne sont pas dans une relation parent-enfant, vous pouvez utiliser un **service singleton** injecté via l'**injection de dépendance (DI)**.

1. **Créer un service Blazor** :

```
public class SharedService
{
    public string Message { get; set; } = "Message global par défaut";
}
```

2. **Enregistrer le service dans** `Program.cs` :

```
builder.Services.AddSingleton<SharedService>();
```

3. **Utiliser le service dans plusieurs composants** :

- **Composant A** :

```
<button @onclick="ChangeMessage">Changer le message</button>

@code {
    [Inject]
    private SharedService SharedService { get; set; }

    private void ChangeMessage()
    {
        SharedService.Message = "Message modifié par Composant A";
    }
}
```

- **Composant B** :

```
<p>Message : @SharedService.Message</p>

@code {
    [Inject]
    private SharedService SharedService { get; set; }
```

Communication via EventAggregator (NuGet)

1. **Installer la bibliothèque `Blazored.EventAggregator` :**

```
dotnet add package Blazored.EventAggregator
```

2. **Enregistrer le service dans `Program.cs` :**

```
builder.Services.AddBlazoredEventAggregator();
```

Communication via EventAggregator (NuGet)

3. **Publier un événement** :

```
[Inject]
private IEventAggregator EventAggregator { get; set; }

private void SendMessage()
{
    EventAggregator.Publish(new CustomEvent { Message = "Hello from A" });
}
```

Communication via EventAggregator (NuGet)

4. **S'abonner à un événement** :

```
[Inject]
private IEventAggregator EventAggregator { get; set; }

protected override void OnInitialized()
{
    EventAggregator.Subscribe(this);
}

public void Handle(CustomEvent customEvent)
{
    Console.WriteLine(customEvent.Message);
}
```

6. Communication par State Container

1. **Créer le conteneur d'état** :

```
public class AppState
{
    public string SharedMessage { get; set; }
}
```

2. **Enregistrer le service dans** `Program.cs` :

```
builder.Services.AddSingleton<AppState>();
```

3. **Utiliser le state dans un composant** :
   - **Composant A** :

```
<input @bind="AppState.SharedMessage" />

@code {
    [Inject]
    private AppState AppState { get; set; }
}
```

6. Communication par State Container

- **Composant B** :

```
<p>Message partagé : @AppState.SharedMessage</p>

@code {
    [Inject]
    private AppState AppState { get; set; }
}
```

Résumé des cas d'usage

Méthode	Type de Communication	Scénario d'Utilisation
[Parameter]	Parent → Enfant	Passer des données simples
EventCallback	Enfant → Parent	Bouton cliqué dans un enfant
Parent → Frères	Via le parent	Enfants communiquent entre eux
Service DI	Global	Partage de données dans l'appli
EventAggregator	Pub/Sub	Découpler les composants
State Container	Partage d’état	État réactif dans l'application

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

SignalR

Introduction à SignalR avec Blazor WebAssembly

Blazor WebAssembly permet d'exécuter du code C# directement dans le navigateur grâce à WebAssembly. Cependant, pour permettre la **communication en temps réel** entre le client (le navigateur) et le serveur, on utilise **SignalR**, qui permet des **communications bidirectionnelles** via WebSockets, Server-Sent Events (SSE) ou Long Polling.

Avec **SignalR**, un serveur peut **envoyer des messages au client à tout moment**, ce qui est utile pour :

- Les notifications en temps réel (exemple : nouvelles alertes).
- Les chats en direct (comme les discussions en ligne).
- Le suivi des événements en direct (suivi d'une commande, suivi des actions utilisateur, etc.).
- Les tableaux de bord en temps réel (mise à jour des métriques).

Introduction à SignalR avec Blazor WebAssembly

1. **Temps réel** : Permet de recevoir des **mises à jour en direct sans recharger la page**.
2. **Événements bidirectionnels** : Le serveur peut **envoyer des messages** au client et le client peut **envoyer des messages** au serveur.
3. **Basé sur WebSocket** : SignalR utilise WebSocket (si disponible), sinon il bascule sur d'autres protocoles comme SSE ou Long Polling.
4. **Facile à utiliser** : Les **Hub Methods** de SignalR facilitent la gestion des messages.
5. **Optimisation des ressources** : Contrairement au "Polling" classique, WebSockets maintient une connexion ouverte, ce qui réduit la charge du serveur.

Architecture d'une application Blazor + SignalR

```
[Blazor WebAssembly (Client)] ↔ [SignalR Hub (Serveur ASP.NET)] ↔ [Base de Données / Back-End]
```

- **Blazor WebAssembly (Client)** : C'est l'application front-end qui interagit avec SignalR pour recevoir et envoyer des messages.
- **SignalR Hub** : Le **hub** se trouve sur le **serveur** (via ASP.NET Core) et permet de centraliser la communication.
- **Base de données / API** : Le hub peut également interagir avec la base de données pour notifier les utilisateurs des modifications de données.

Configuration du projet

Étape 1 : Création du projet

1. **Créer un projet Blazor WebAssembly (ASP.NET Core hébergé)**

```
dotnet new blazorwasm --hosted -o BlazorSignalRApp
```

Ce modèle inclut :

- **Client** (Blazor WebAssembly) — Vue client.
- **Server** (ASP.NET Core) — Pour le hub SignalR.
- **Shared** — Pour partager des types de données entre le serveur et le client.

Configuration du projet

Étape 2 : Ajouter SignalR au projet

1. **Installer le package NuGet dans le projet Serveur** :

```
dotnet add package Microsoft.AspNetCore.SignalR
```

2. **Ajouter le hub dans le projet Serveur** :
   - Créez un fichier `ChatHub.cs` dans le projet **Server**.

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

public class ChatHub : Hub
{
    // Envoie un message à tous les clients connectés
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", user, message);
    }
}
```

Configuration du projet

3. **Configurer le hub dans le `Program.cs` du serveur** :

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapControllers();
app.MapBlazorHub();
app.MapFallbackToPage("/_Host");

// Ajout de SignalR
app.MapHub<ChatHub>("/chathub");

app.Run();
```

Configuration du projet

Étape 3 : Configurer le client Blazor WebAssembly

1. **Installer le package SignalR dans le projet Client** :

```
dotnet add package Microsoft.AspNetCore.SignalR.Client
```

Configuration du projet

2. **Configurer la connexion au hub SignalR** :
   - Ouvrez le fichier `Program.cs` dans le projet **Client** et ajoutez le service de SignalR.

```
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;
using Microsoft.Extensions.DependencyInjection;

var builder = WebAssemblyHostBuilder.CreateDefault(args);
builder.RootComponents.Add<App>("#app");

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

// Ajouter SignalR
builder.Services.AddScoped<HubConnection>(sp =>
    new HubConnectionBuilder()
        .WithUrl($"{builder.HostEnvironment.BaseAddress}chathub")
        .Build()
);

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

Configuration du projet

Étape 4 : Intégrer SignalR dans une page Blazor

1. **Créer un fichier `Chat.razor` dans le dossier Pages :**

```
@page "/chat"
@inject HubConnection HubConnection

<h3>Chat en direct</h3>

<input @bind="UserName" placeholder="Nom d'utilisateur" />
<input @bind="Message" placeholder="Message" />
<button @onclick="SendMessage">Envoyer</button>

<ul>
    @foreach (var msg in Messages)
    {
        <li><strong>@msg.User:</strong> @msg.Message</li>
    }
</ul>

@code {
    private string UserName;
    private string Message;
    private List<(string User, string Message)> Messages = new List<(string, string)>();

    protected override async Task OnInitializedAsync()
    {
        // Se connecter au hub
        HubConnection.On<string, string>("ReceiveMessage", (user, message) =>
        {
            Messages.Add((user, message));
            InvokeAsync(StateHasChanged);
        });

        await HubConnection.StartAsync();
```

Intérêt de SignalR dans Blazor WebAssembly

Fonctionnalité	Utilité
Chat en direct	Discussions en temps réel.
Notifications	Notifications système.
Jeux en ligne	Mise à jour des jeux multi-joueurs.
Tableaux de bord	Indicateurs qui se mettent à jour.
Suivi des commandes	Visualiser l'avancement des commandes.
Support en ligne	Fenêtre de support instantané.

Optimisations et bonnes pratiques

1. **Utiliser des groupes** :
   - Vous pouvez regrouper les utilisateurs dans des **Groupes**. Par exemple, vous pouvez envoyer des notifications à des groupes d'utilisateurs spécifiques :

```
public async Task JoinGroup(string groupName)
{
    await Groups.AddToGroupAsync(Context.ConnectionId, groupName);
}
```

2. **Authentification et sécurité** :
   - Protégez les routes SignalR avec l'authentification **JWT** ou **ASP.NET Identity**.
3. **Limiter le nombre de clients** :
   - Les clients ne doivent pas recevoir trop de messages à la fois. Préférez **le broadcast différé**.

Optimisations et bonnes pratiques

4. **Gestion des déconnexions** :
    - Vous pouvez suivre quand un utilisateur se connecte ou se déconnecte :

```
public override async Task OnDisconnectedAsync(Exception? exception)
{
    await base.OnDisconnectedAsync(exception);
}
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Interaction avec DOM

Techniques de partage de code et interaction avec le DOM

Blazor permet une intégration transparente de JavaScript dans vos applications, facilitant ainsi l'utilisation des bibliothèques existantes.

1. Pourquoi devez-vous utiliser JavaScript dans un projet Blazor ?

Il existe des scénarios dans lesquels il peut être nécessaire ou bénéfique d'utiliser JavaScript dans un projet Blazor. Voici quelques raisons pour lesquelles vous pourriez avoir besoin d'utiliser JavaScript dans un projet Blazor :

- Interagir avec le DOM du navigateur : Blazor fournit un riche ensemble de composants UI qui peuvent être utilisés pour construire des applications web, mais il peut y avoir des cas où vous devez interagir directement avec le modèle objet du document (DOM) du navigateur pour atteindre une fonctionnalité spécifique. Par exemple, créer un événement HTML personnalisé, gérer un événement HTML fréquemment déclenché comme mousemove. Dans ces cas, vous pouvez utiliser JavaScript pour gérer cela afin d'augmenter la performance ou créer un événement HTML personnalisé.
- Accéder aux API du navigateur : Blazor permet d'accéder à une large gamme d'API .NET qui peuvent être utilisées pour interagir avec le navigateur et d'autres ressources système. Cependant, il peut y avoir des cas où vous devez accéder aux API du navigateur qui ne sont actuellement pas disponibles en .NET. Par exemple, accéder à la caméra du navigateur, utiliser IndexedDb. Dans ces cas, vous pouvez utiliser JavaScript pour accéder à l'API du navigateur et appeler des fonctions JavaScript depuis le code C#.
- Bibliothèques tierces : Il peut y avoir des cas où vous souhaitez utiliser une bibliothèque JavaScript tierce qui n'est pas disponible en .NET (encore). Dans ces cas, vous pouvez utiliser JavaScript pour inclure la bibliothèque dans votre projet et appeler ses fonctions depuis le code C#.

Techniques de partage de code et interaction avec le DOM

2. JavaScript standard et module JavaScript

Il existe 2 manières d'implémenter JavaScript : le JavaScript standard et le module JavaScript.

- Le JavaScript standard fait référence aux fonctionnalités et fonctionnalités de base du langage définies dans la spécification ECMAScript, qui est la norme officielle pour le langage JavaScript.
- Les modules JavaScript, en revanche, sont un moyen d'organiser et de structurer le code JavaScript en composants réutilisables qui peuvent être importés et exportés entre fichiers.

Techniques de partage de code et interaction avec le DOM

3. Communication entre les environnements .NET et JavaScript

- IJSRuntime est disponible à la fois pour Blazor Server et Blazor WebAssembly et est généralement utilisé à l'intérieur d'un composant. Il peut également être utilisé pour modulariser le code JavaScript. Lorsque vous utilisez IJSRuntime, vous devez contrôler la durée de vie du module JavaScript et de ses références.
- JSImport/JSExport est exclusivement disponible pour Blazor WebAssembly et est utilisé pour modulariser le code JavaScript. Cette fonctionnalité offre un contrôle plus strict sur le processus de marshalling des données en définissant les paramètres et les types de retour des fonctions JavaScript en tant que leurs équivalents de types de données C#. Cela empêche des données de retour imprévisibles des fonctions JavaScript. Importamment, l'utilisation de JSImport/JSExport élimine le besoin de gérer la durée de vie du module et de ses références.

Techniques de partage de code et interaction avec le DOM

Lorsque vous développez des applications web avec Blazor, vous aurez souvent besoin de stocker des données localement sur l'appareil de l'utilisateur. C'est là que le stockage du navigateur s'avère utile, offrant un moyen de stocker et de récupérer des données directement dans le navigateur de l'utilisateur. Dans cette vue d'ensemble, nous explorerons les différents types de stockages disponibles dans Blazor et comment les utiliser efficacement dans vos applications.

1. L'importance de l'utilisation du stockage du navigateur

Le stockage du navigateur est une fonctionnalité essentielle lors du développement d'applications web car il vous permet de stocker des données sur l'appareil de l'utilisateur, qui peuvent être rapidement et facilement accessibles par votre application. Ceci est particulièrement important lorsqu'on travaille avec des données qui doivent être persistantes, telles que les préférences utilisateur, les identifiants de connexion ou l'état de l'application.

Techniques de partage de code et interaction avec le DOM

2. Types de stockage du navigateur

- **Cache :** Ce type de stockage est utilisé pour stocker des données temporaires, telles que des images ou des scripts, qui sont fréquemment accédés par votre application. Le cache aide à améliorer les performances en permettant à votre application de charger rapidement les données sans avoir à faire une requête serveur à chaque fois.
- **Cookies :** Les cookies sont de petits morceaux de données qui sont stockés sur l'appareil de l'utilisateur par le navigateur. Ils sont souvent utilisés pour stocker les préférences des utilisateurs, les données de session ou les identifiants de connexion. Les cookies sont envoyés au serveur à chaque requête, permettant à votre application de se souvenir des paramètres de l'utilisateur ou de maintenir les utilisateurs connectés.
- **IndexedDB :** IndexedDB est une base de données côté client qui vous permet de stocker de grandes quantités de données structurées sur l'appareil de l'utilisateur. Ce type de stockage est utile lorsqu'on travaille avec des applications web hors ligne ou lorsqu'il est nécessaire de stocker de grandes quantités de données qui seraient lentes à récupérer depuis le serveur.
- **Mémoire :** peut être utile pour stocker des données qui doivent être rapidement accessibles et mises à jour par l'application, telles que les caches en mémoire ou les variables utilisées pour les calculs. Cependant, il est important de noter que les données stockées en mémoire sont volatiles, ce qui signifie qu'elles seront perdues lorsque la page sera rafraîchie ou fermée.

Techniques de partage de code et interaction avec le DOM

- **Stockage local :** Le stockage local est similaire aux cookies mais peut stocker de plus grandes quantités de données. Ce type de stockage est souvent utilisé pour stocker les paramètres utilisateur, l'état de l'application ou des données qui doivent persister entre les sessions.
- **Stockage de session :** Le stockage de session est similaire au stockage local, mais les données sont effacées lorsque l'utilisateur ferme la fenêtre du navigateur. Ce type de stockage est utile lorsqu'on travaille avec des données temporaires ou lorsqu'on ne souhaite pas que les données persistent entre les sessions.

Techniques de partage de code et interaction avec le DOM

Une comparaison des différents types de stockage du navigateur

Type de stockage	Durée de vie	Type de données autorisées	Partagé entre les onglets
Cache	Jusqu'à suppression	Requête, réponse	Oui
Cookie	Durée expirée/jusqu'à suppression	Clé-valeur	Oui
IndexedDB	Jusqu'à suppression	Types variés	Oui
Stockage local	Jusqu'à suppression	Clé-valeur	Oui
Mémoire	Lorsque l'utilisateur	quitte l'onglet	Types variés
Stockage de session	Lorsque l'utilisateur quitte l'onglet	Clé-valeur	Non

Authentification, autorisation et sécurisation des applications Blazor.

L'authentification et l'autorisation jouent un rôle vital dans le maintien de la sécurité et de la confidentialité des applications web et de leurs utilisateurs dans le développement web. En d'autres termes, la mise en œuvre de l'authentification et de l'autorisation vous permet d'afficher une interface utilisateur unique pour chaque utilisateur en fonction de son rôle.

1. Comment fonctionne l'authentification dans Blazor ?

Pour comprendre comment fonctionne l'authentification dans Blazor, vous devez connaître AuthenticationStateProvider et CascadingAuthenticationState, ainsi que comment utiliser le stockage du navigateur pour stocker les identifiants des utilisateurs. De plus, il est important de comprendre les 3 flux principaux d'identification, qui incluent le flux de connexion, le flux de revisite du site web par l'utilisateur, et le flux de déconnexion.

Comprendre les éléments de l'authentification

La mise en œuvre de l'authentification dans Blazor nécessite plusieurs éléments clés. Ceux-ci incluent :

- Stockage des données du navigateur
- AuthenticationStateProvider
- CascadingAuthenticationState

Authentification, autorisation et sécurisation des applications Blazor.

- Le stockage des données du navigateur peut être réalisé via la mémoire, le stockage local ou le stockage de session. Ce stockage est responsable du maintien des données de l'utilisateur et assure qu'elles persistent même lorsque l'utilisateur ouvre un nouvel onglet ou rafraîchit la page. Le stockage local est recommandé à cet effet.
- L'AuthenticationStateProvider est responsable de la gestion de l'authentification des utilisateurs, déterminant si un utilisateur est connecté ou non. Il contient votre logique métier d'authentification et est responsable de notifier les autres éléments sur le statut d'authentification de l'utilisateur.
- CascadingAuthenticationState est responsable de la transmission de l'état d'authentification à travers le site web. En tant que composant racine, il fonctionne de manière transparente en arrière-plan, et il n'est pas nécessaire de le modifier ou de le configurer.

Authentification, autorisation et sécurisation des applications Blazor.

2. Comment fonctionne l'autorisation dans Blazor ?

Pour comprendre comment fonctionne l'autorisation dans Blazor, il est essentiel de connaître le modèle Identity, qui comprend ClaimsPrincipal, ClaimsIdentity et Claim. De plus, vous devez être familiarisé avec les règles d'autorisation et les métadonnées des ressources autorisées.

Le modèle Identity

Le modèle Identity est un cadre qui fournit un ensemble de classes et d'interfaces pour gérer l'authentification et l'autorisation des utilisateurs dans les applications .NET, y compris Blazor. Le modèle Identity définit une manière standard de représenter et de gérer les informations d'identité de l'utilisateur, y compris leur ID utilisateur, nom d'utilisateur, mot de passe et toute information supplémentaire ou revendications qui peuvent être associées à leur identité.

Les éléments clés du modèle Identity incluent :

- ClaimsPrincipal : Représente l'identité de l'utilisateur et contient une collection de revendications, qui sont des informations sur l'utilisateur telles que leur nom, rôle ou adresse email.
- ClaimsIdentity : Représente un ensemble de revendications associées à une identité spécifique et fournit des méthodes pour ajouter ou supprimer des revendications.
- Claim : Représente une seule information sur l'utilisateur, telle que leur nom, email ou rôle.

Authentification, autorisation et sécurisation des applications Blazor.

Règle d'autorisation

Une règle d'autorisation consiste en un ensemble de conditions qu'un utilisateur doit remplir pour accéder à une ressource protégée. Ces conditions peuvent inclure des exigences telles qu'appartenir à un groupe spécifique, avoir un rôle particulier ou répondre à certains critères d'âge. Si un utilisateur ne satisfait pas à toutes les conditions de la règle d'autorisation, il se verra refuser l'accès à la ressource.

Authentification, autorisation et sécurisation des applications Blazor.

3. Métadonnées des ressources d'autorisation

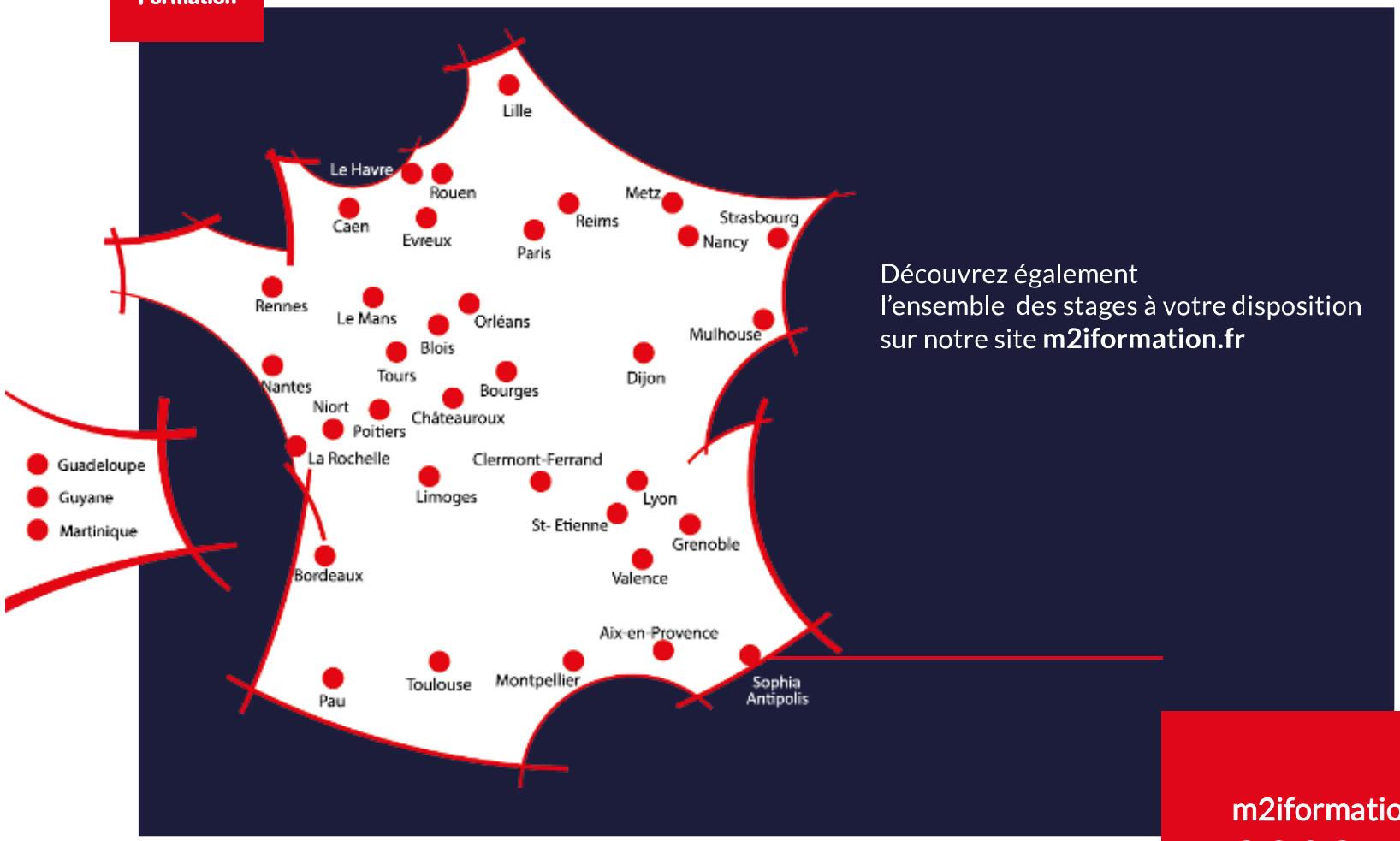
Les métadonnées des ressources d'autorisation se réfèrent aux données ou informations spécifiques associées à une ressource qui sont utilisées pour prendre des décisions d'autorisation. Ces métadonnées peuvent inclure divers attributs, tels que le rôle de l'utilisateur, les permissions et autres informations pertinentes. Lorsqu'un utilisateur tente d'accéder à une ressource protégée, le système d'autorisation vérifie les informations d'identification de l'utilisateur et les compare aux métadonnées associées à la ressource.

Par exemple, considérons un magasin de proximité qui vend divers produits. L'un des produits est l'alcool, qui a une restriction d'âge de 18 ans ou plus. D'autres produits peuvent avoir différentes restrictions d'âge ou pas de restrictions du tout. Dans ce cas, la condition d'âge de 18 ans est la métadonnée de ressource autorisante utilisée pour contrôler l'accès au produit alcoolisé.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Merci pour votre attention

Des questions ?



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr