

Blazor WebAssembly

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 {  
    [Parameter]  
    public string UserId { get; set; }  
}
```

Routing avec Blazor

3. Navigation

Pour naviguer entre les pages, Blazor fournit le composant `NavLink` qui génère un lien HTML (``). 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.

Blazor Server

Blazor Server fonctionne en exécutant le code C# sur le serveur au sein d'une session ASP.NET Core. L'interface utilisateur et les événements du navigateur sont gérés via une connexion SignalR, un framework pour ajouter des fonctionnalités de communication en temps réel à des applications web. Cette connexion permet un échange dynamique de données et d'actions entre le navigateur et le serveur. L'état de l'application est conservé sur le serveur, ce qui réduit la charge sur le client mais nécessite une connexion persistante et rapide pour une expérience utilisateur fluide.

SignalR

SignalR est une bibliothèque open source pour ASP.NET qui permet de développer des applications web nécessitant des fonctionnalités en temps réel. Grâce à SignalR, les serveurs peuvent envoyer du contenu aux navigateurs et aux clients connectés instantanément, sans que le client n'ait besoin de sonder le serveur pour obtenir des mises à jour. SignalR est particulièrement utile pour les applications nécessitant des interactions en temps réel, comme les jeux en ligne, les réseaux sociaux, les plateformes de trading, et les applications de chat.

1. Comment ça fonctionne ?

SignalR utilise plusieurs techniques de transport pour établir une communication en temps réel entre le client et le serveur. Il sélectionne automatiquement la meilleure technique disponible basée sur les capacités du serveur et du client. Ces techniques incluent :

- **WebSockets** : C'est le mode de transport privilégié si les deux, client et serveur, le prennent en charge. WebSockets permet une communication bidirectionnelle pleinement interactive entre le client et le serveur.
- **Server-Sent Events** : Utilisé principalement par les navigateurs clients pour recevoir des mises à jour automatiques du serveur.
- **Long Polling** : Une technique de secours si les autres ne sont pas disponibles. Dans le long polling, le client envoie une requête au serveur, qui reste ouverte jusqu'à ce que le serveur puisse envoyer des données au client.

SignalR

- **Gestion des connexions** : SignalR gère automatiquement la gestion des connexions, y compris la reconnexion automatique en cas de perte de connexion.
- **Diffusion en temps réel** : Les serveurs peuvent diffuser des messages à tous les clients connectés simultanément ou à des clients spécifiques.
- **Groupes** : Les clients peuvent être ajoutés à des groupes spécifiques, permettant ainsi au serveur d'envoyer des messages à des groupes de clients plutôt qu'à des clients individuels ou à tous les clients.
- **Interopérabilité** : SignalR peut être utilisé avec différents types de clients, pas seulement les navigateurs web, mais aussi des applications .NET, des applications Java, etc.

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 marshallage 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.

Merci pour votre attention

Des questions ?

