Chapitre 4
Fonctionnalités avancées

1. JavaScript et CSS

Maintenant que les composants ont été explorés, il est temps de voir en détail les fonctionnalités avancées de ces derniers. Nous allons commencer par traiter les fonctionnalités qui concernent JavaScript et CSS, pour ensuite aborder les ajouts faits en .NET 5.

1.1 Interopérabilité avec JavaScript

Même si Blazor permet d'écrire du code C# à la place de JavaScript pour les interactions les plus communes, il n'en reste pas moins qu'il existe une panoplie de composants et de projets JavaScript créés par la communauté pour répondre à un vaste ensemble de besoins. Blazor permet de travailler avec ces derniers, et ce de manière bidirectionnelle (c'est-à-dire qu'il est possible d'intégrer un composant JavaScript et de lui envoyer des données, mais aussi d'exécuter du code C# depuis JavaScript). Nous allons voir ces deux modes.

1.1.1 Invocation JavaScript depuis C#

C'est le scénario le plus courant. Il existe bien des cas d'usage où il est nécessaire d'avoir recours à un composant JavaScript et donc d'appeler des méthodes JavaScript depuis le code C#. L'équipe de développement de Blazor a prévu ces cas : elle a mis à disposition une interface, `IJSRuntime`, permettant d'effectuer cette manipulation. Cette interface est directement utilisable par injection de dépendances dans les composants :

```
[Inject]
public IJSRuntime JSRuntime{ get; set; }
```

Comme cela a été évoqué dans le chapitre Les composants en détail, il est nécessaire de réaliser les appels JavaScript à un niveau assez tardif dans le cycle de vie d'un composant, c'est-à-dire au minimum dans la méthode `OnAfterRenderAsync`. Cela permet d'assurer que la totalité du composant et de la communication est disponible afin de pouvoir traiter l'ordre en JavaScript. Il y a deux façons distinctes d'appeler une méthode JavaScript :

– Invocation d'une méthode qui ne renvoie aucun paramètre. Cela se passe avec la méthode `InvokeVoidAsync`.
– Invocation d'une méthode qui renvoie un paramètre. Cela se passe avec la méthode `InvokeAsync`.

Par exemple, si nous souhaitons afficher une alerte à l'aide de la méthode JavaScript `alert` lorsque le compteur est proche de 10, nous utilisons la méthode `InvokeVoidAsync`, car l'affichage d'une alerte ne nécessite aucun retour de paramètre. À l'inverse, si nous souhaitons demander à l'utilisateur, à l'aide de la méthode `prompt` la valeur initiale du compteur, nous utilisons la méthode `InvokeAsync`. Procédons à cela à titre d'exercice. Le runtime JavaScript étant injecté grâce au code vu précédemment, nous allons écrire les méthodes JavaScript permettant de réaliser ces deux opérations.

Pour que ces méthodes puissent être invoquées depuis C#, il faut qu'elles soient accessibles. En effet, il n'est pas possible de mettre du code JavaScript dans un composant Blazor. Le seul point d'entrée disponible au niveau de notre application se trouve de fait au niveau de notre _Host.cshtml. De la même façon, il est préférable d'avoir une référence vers le périmètre global disponible en JavaScript, et cela passe par l'objet `window`.

JavaScript n'étant pas un langage orienté objets par essence, il est possible d'ajouter dynamiquement des fonctions et des données à un objet existant sans pour autant que cela cause d'erreur. Nous allons de ce fait créer nos deux méthodes :

```
<script type="text/javascript">
    window.showAlert = (number) => {
        alert("Attention, vous êtes actuellement à
" + number + ". A 10, le compteur sera bloqué.");
    };
    window.promptUser = () => {
        let result = prompt("Saisissez la valeur initiale du
compteur : ");
        return result;
    };
</script>
```

En créant le code suivant, nos méthodes sont désormais disponibles à un niveau global, et nous pouvons les appeler depuis notre code C#. Ainsi, nous transformons la méthode `IncrementCount` afin de pouvoir afficher l'alerte au besoin :

```
public async Task IncrementCount(MouseEventArgs e)
{
    if (CurrentCount >= 10)
    {
        return;
   }

    if (e.AltKey)
    {
        CurrentCount += 2;
    }
    else
    {
        CurrentCount++;
    }
    if (Math.Abs(10 - CurrentCount) <= 2)
    {
        await JSRuntime.InvokeVoidAsync("showAlert", CurrentCount);
    }
}
```

Comme on peut le constater, l'appel à la méthode `InvokeVoidAsync`, en passant en premier paramètre le nom de notre fonction, n'est possible que parce que la fonction est déclarée sur la portée globale à l'aide de l'objet `window`.

Remarque

JavaScript offre une portée globale à l'aide du mot-clé var. Il est ainsi possible de créer un objet global, à l'aide de var, et d'y ajouter nos fonctions. L'appel de ces fonctions nécessite de préfixer l'appel à la méthode par le nom de la variable correspondant à cet objet global. Côté JavaScript, cela donne : `var myFunctions = myFunctions || {}; myFunctions.showAlert = (number) => { ... }`. Et côté C# : `JSRuntime.InvokeVoidAsync("myFunctions.showAlert");`

Voici le code pour demander à l'utilisateur de saisir la valeur initiale du compteur à l'aide de la méthode `prompt` :

```
protected override async Task OnAfterRenderAsync(bool firstRender)
 {
     if (firstRender)
     {
         var initial =
await JSRuntime.InvokeAsync<string>("promptUser");
         if (int.TryParse(initial, out int initVal))
         {
             CurrentCount = initVal;
             await InvokeAsync(StateHasChanged);
         }
    }
    await base.OnAfterRenderAsync(firstRender);
}
```

Comme cela a été dit, il est possible de réaliser cet appel au plus tôt dans `OnAfterRenderAsync`, et pas avant. Nous nous servons donc du paramètre `firstRender`, afin de déterminer s'il s'agit de l'affichage initial ou d'une mise à jour de l'affichage (si nous le faisions de façon systématique en dehors de l'affichage initial, nous entrerions dans une boucle sans fin où nous demanderions la valeur à l'utilisateur de façon systématique). Afin de nous assurer que l'interface est bien mise à jour, une fois que nous avons affecté la valeur `CurrentCount`, nous appelons la méthode `StateHasChanged` pour notifier la bonne prise en compte du changement d'état du composant. Ainsi, l'utilisateur doit saisir la valeur initiale uniquement au premier affichage.

Remarque

Lors des échanges entre C# et JavaScript, il est préférable d'éviter de faire transiter des objets ou des valeurs autres qu'une chaîne de caractères pour éviter d'éventuelles erreurs de conversion. Ainsi, dans l'exemple précédent, on récupère une valeur de type chaîne de caractères et on s'assure que l'utilisateur a saisi une valeur numérique (rien n'empêche de modifier ce comportement pour forcer l'utilisateur à saisir une valeur numérique en boucle). Mais si on met `int` à la place de `string` dans le paramètre générique précédent, on a une erreur de conversion, car JavaScript renvoie une chaîne au format JSON et C# ne fait pas la conversion de façon automatique. Pour plus de robustesse, il est préférable de coder ces transformations soi-même.

1.1.2 Invocation C# depuis JavaScript

Jusqu'à présent, nous avons invoqué du JavaScript depuis C#. Blazor permet également d'effectuer l'opération inverse : exécuter une fonction C# depuis du code JavaScript. Cela s'avère très pratique lors de l'utilisation de packages communautaires qui doivent mettre à jour quelque chose du côté back-end. Généralement, on passe par une notion d'appel AJAX avec les services web. Avec Blazor, ceci n'est plus nécessaire.

En guise d'exemple, nous allons créer un bouton spécial qui incrémente le compteur de 3 en 3, et utiliser JavaScript pour appeler la méthode C# concernée. Ce comportement est bien sûr possible avec Blazor sans JavaScript, mais le but est ici de démontrer comment on peut appeler notre code C# depuis le code JavaScript.

La toute première étape est de créer la fonction qui effectue cette opération, en C#, et de lui ajouter l'attribut `JSInvokable` :

```
[JSInvokable]
public async Task IncrementByThree()
{
    CurrentCount += 3;
    await InvokeAsync(StateHasChanged);
}
```

Maintenant que cette méthode est créée, nous ajoutons le code HTML du bouton concerné :

```
<button class="btn btn-primary" onclick="Increment3()">
Increment 3</button>
```

On utilise bien cette fois l'événement `onClick` HTML qui se connecte à du JavaScript du fait de l'absence du préfixe `@`.

Il nous reste à écrire la fonction JavaScript `Increment3`. Dans cette fonction, on doit avoir la possibilité d'invoquer notre composant. On a donc besoin d'une référence d'objet vers ce dernier. Il est donc nécessaire, lorsque le composant s'initialise, qu'il crée une référence vers lui-même afin que JavaScript sache comment l'invoquer.

Au niveau de notre composant, toujours dans la méthode `OnAfterRenderAsync` (vu que l'on interagit avec JavaScript, ce devrait maintenant être un réflexe), on crée cette référence avec la classe `DotNetObjectReference`. Une fois cette référence obtenue, il faut la transmettre au code JavaScript afin qu'il la stocke. Pour faire cette opération, il faut invoquer une méthode JavaScript en lui passant l'objet ainsi créé :

```
protected override async Task OnAfterRenderAsync(bool firstRender)
{
    if (firstRender)
    {
        ...
        var thisRef = DotNetObjectReference.Create(this);
        await
JSRuntime.InvokeVoidAsync("storeRazorCounterComponent", thisRef)
    }
    await base.OnAfterRenderAsync(firstRender);
}
```

Côté JavaScript, il ne nous reste plus qu'à coder les méthodes `storeRazorCounterComponent` et `Increment3`. La première stocke la référence vers le composant Blazor dans une variable globale, et la seconde vérifie si cette variable globale existe. Si tel est le cas, elle appelle la méthode `invokeMethodAsync` en passant en paramètre le nom de la méthode à invoquer côté Blazor. Tout ceci s'ajoute dans notre _Host.cshtml :

```
var counterComponent;

window.storeRazorCounterComponent = (ref) => { counterComponent = ref; };
function Increment3() {
    if (counterComponent) {
        counterComponent.invokeMethodAsync('IncrementByThree');
    }
};
```

Maintenant, notre nouveau bouton appelle bien notre méthode Blazor de façon transparente, ce qui correspond, à peu de choses près, à un appel de méthode AJAX.

1.2 Nouveautés .NET 5

.NET 5 a apporté son lot de fonctionnalités avancées, et nous allons voir dans cette section quelques ajouts intéressants.

1.2.1 Isolation CSS

Lorsque l'on crée un composant, il est fort probable qu'à un moment ou à un autre, on en arrive à avoir besoin d'éditer le code CSS de ce dernier. Souvent, cela se traduit par l'ajout d'une ou de plusieurs nouvelles classes CSS. Il est possible d'utiliser des outils, comme gulp, qui rassemblent en bundle la totalité des fichiers CSS au moment de la compilation pour ne produire qu'un seul fichier. Cela nous permet de créer un fichier CSS par bloc logique.

Avec Blazor en .NET 5, il est possible d'utiliser une nouvelle technique : l'isolation CSS. Cela permet d'avoir un fichier CSS par composant, avec un style qui ne s'applique qu'au composant concerné. Grâce à cette pratique, il est possible d'utiliser des styles globaux dans le fichier CSS (par exemple pour définir le style de tous les boutons HTML, représentés par l'élément `<button>`), mais cela est limité au composant.

Concrètement, il suffit de réaliser deux actions :

– Créer un fichier CSS qui porte le nom du composant. Si notre composant est dans le fichier Counter.razor, il faut créer Counter.razor.css.

– Ajouter dans la balise `<head>` de notre application le lien vers le fichier de styles qui est généré par Blazor. Ce fichier porte le nom du projet suivi de .Styles.css.