Chapitre 4
Algorithmique

1. Bases de l'algorithmique

Jusqu'à présent, nous nous sommes contentés de développer des applications n'incluant aucune "logique" : elles se limitaient à afficher des données. Il n'y avait aucune notion de condition, de répétition ou même de logique de code. En effet, le code d'une application est souvent complexe et les embranchements sont multiples en fonction de diverses conditions. Dans ce chapitre, nous allons découvrir la logique algorithmique, qui vous permettra de créer du code plus proche de ce que l'on peut retrouver dans les applications répondant à des problématiques plus complexes.

1.1 La logique conditionnelle

Indéniablement, il s'agit ici d'une brique que vous allez utiliser de façon systématique. Une condition implique l'exécution ou non d'une partie du code en fonction de l'évaluation d'un test logique.

1.1.1 Test simple : le if/else

La logique conditionnelle se traduit en pseudocode de la façon suivante :

```
SI une condition ALORS
  Je fais quelque chose
SINON
  Je fais autre chose
```

En C#, les mots-clés pour réaliser une instruction conditionnelle sont `if` et `else` :

```
if(condition)
{
    ....
}
else
{
    ....
}
```

La condition testée par une instruction `if` doit renvoyer un booléen. Ce dernier peut être stocké dans une variable mais il est également possible que l'instruction `if` évalue directement la condition, sans variable intermédiaire.

Si on reprend l'exemple de la fin du chapitre précédent, on pourrait améliorer notre classe `Voiture` pour rajouter un booléen qui indique si l'instance de la voiture est fonctionnelle. Si la valeur est égale à "oui", il est inutile de réparer la voiture. Cependant, si la voiture n'est pas fonctionnelle, il faut la réparer :

```
public class Voiture
{
    public bool Fonctionnelle { get; set; }
    ...
}
public class Garage
{
    public void Repare(Voiture voiture)
    {
        if(voiture.Fonctionnelle)
        {
              Console.WriteLine("La voiture n'a pas besoin d'être
réparée car elle est fonctionnelle");
        }
```

```
        else
        {
              Console.WriteLine("Réparation de la voiture");
              voiture.Fonctionnelle = true;
        }
    }
}
```

Comme on le voit dans le code ci-dessus, l'instruction `if` se base sur la valeur booléenne stockée dans la propriété `Fonctionnelle` de la classe `voiture` pour évaluer si oui ou non la réparation est nécessaire. Ici, le test a été fait de telle sorte que l'on vérifie si la condition est vraie, et dans le cas inverse, on effectue la réparation. On peut très bien inverser la condition initiale, en comparant le booléen à la valeur `false`. De ce fait, on peut même se passer du `else`, qui n'apporte pas réellement de plus-value :

```
public void Repare(Voiture voiture)
    {
        if(voiture.Fonctionnelle == false)
        {
              Console.WriteLine("Réparation de la voiture");
              voiture.Fonctionnelle = true;
        }
    }
```

À noter également qu'il est possible d'inverser la valeur d'un booléen en mettant un point d'exclamation en préfixe. Ainsi, `!true` est égal à `false`, et `!false` est égal à `true`. Même si cela peut sembler compliqué de prime abord, vous verrez que c'est une façon d'écrire qui deviendra rapidement automatique à l'utilisation. Si l'on reprend l'exemple précédent, le code qui utilise l'inversion de valeur avec le point d'exclamation serait le suivant :

```
public void Repare(Voiture voiture)
    {
        if(!voiture.Fonctionnelle)
        {
              Console.WriteLine("Réparation de la voiture");
              voiture.Fonctionnelle = true;
        }
    }
```

Même si à première vue l'instruction `else` est utilisée pour définir le cas inverse de celui du `if` principal, elle peut également servir de base pour une autre instruction `if` à suivre afin de faire une instruction ayant pour sémantique "sinon si". Il suffit dans ce cas d'ajouter une condition `if` après le `else`. Par exemple :

```
public void DecrireVoiture(Voiture voiture)
{
    if(voiture.Marque == "Ferrari")
    {
        Console.WriteLine("Voiture chère");
    }
    else if(voiture.Marque == "Peugeot")
    {
        Console.WriteLine("Voiture standard");
    }
    else
    {
         Console.WriteLine("Marque de voiture non reconnue");
    }
}
```

À noter que la structure du code conditionnel est très flexible : on peut avoir uniquement une seule instruction `if`, une instruction `if` et son `else` associé, ou un enchaînement de `if` et `else if` (avec ou sans `else` final). La seule impossibilité : avoir une instruction `else` seule, car cette dernière indique forcément l'inverse d'une condition donnée.

Remarque

Pour que ces embranchements soient possibles, il faut bien sûr qu'il y ait des conditions pouvant donner plusieurs résultats. À ce titre, il n'est pas utile de faire un `if`, `else if`, `else` avec un simple booléen, car ce dernier ne pouvant avoir que deux états, un `if` avec un `else` est suffisant.

Une instruction `if` peut être "compressée" en l'exprimant sous une forme réduite appelée ternaire. Généralement, on utilise cette approche afin d'écrire en ligne un test pour éviter une lourdeur syntaxique, et ce afin d'affecter le contenu d'une variable. La syntaxe est la suivante : on définit en première partie le test à évaluer, séparant d'un point d'interrogation le test des résultats. Ensuite, les cas vrai et faux sont tous deux séparés par un deux-points. La syntaxe est la suivante :

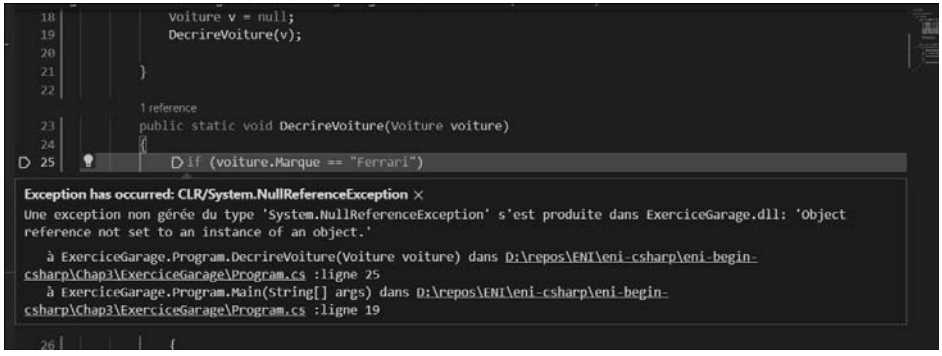
```
test ? cas si vrai : cas si faux
```

Par exemple :

```
string voiture = voiture.Marque == "Ferrari" ? "Voiture chère" :
"Voiture peu chère";
```

Il est possible d'enchaîner les ternaires avec l'usage des parenthèses (en refaisant une autre ternaire dans un cas ou l'autre) mais il est recommandé d'agir avec parcimonie afin de conserver une lisibilité de code optimale.

Il est souvent intéressant de tester si un objet a été affecté avant d'accéder à ses données ou à ses méthodes. En l'absence de ce test, cela peut provoquer une erreur à l'exécution (appelée exception, que nous détaillerons dans ce chapitre à la section La gestion des erreurs). Si on reprend le code précédent, étant donné que `Voiture` est une classe, elle peut donc valoir la valeur `null`. La fonction `DecrireVoiture` tenterait dès lors d'accéder à une variable qui n'a pas de valeur, provoquant une erreur à l'exécution :



Erreur à l'exécution

Afin d'effectuer un quelconque test sur une donnée d'une classe ou de faire un appel de méthode, il est recommandé de tester si la valeur est bien différente de `null`. Cela peut se faire de façon "classique" ou grâce au nouvel apport du mot-clé `not` de C# 9 (comme cela est décrit dans la section à venir Pattern matching) :

```
public void DecrireVoiture(Voiture voiture)
{
    if (voiture != null) // avant C# 9
    {
          ...
    }
    if (voiture is not null) // depuis C# 9
    {
          ...
    }
}
```

Pour éviter ce genre de problèmes, un opérateur de navigation sécurisé a été ajouté en C# 6. Ce dernier permet de n'accéder à une méthode ou de lire une donnée que si la variable n'est pas `null`. On utilise le point d'interrogation juste après la variable, avant l'appel, et cela permet de se passer de tester la nullité :

```
public void DecrireVoiture(Voiture voiture)
{
    if(voiture?.Marque == "Ferrari")
    {
        Console.WriteLine("Voiture chère");
    }
    else if(voiture?.Marque == "Peugeot")
    {
        Console.WriteLine("Voiture standard");
    }
    else
    {
         Console.WriteLine("Marque de voiture non reconnue");
    }
}
```

Le fonctionnement de cet opérateur est le suivant :

- Si la variable n'est pas `null`, on accède à la propriété ou à la méthode concernée normalement.
- Si la variable est `null` :
  - S'il s'agit d'un appel d'une méthode, et que la méthode ne renvoie rien, elle ne sera pas invoquée ;
  - S'il s'agit d'un appel d'une méthode et que la méthode renvoie une valeur, ou qu'il s'agit d'un appel à une propriété, il faudrait tester si la valeur est différente de `null`. S'il s'agit d'un type référence (d'une classe, comme un `string`), alors il faudrait tester si c'est `null` ou pas, afin de voir si l'appel a été fait. S'il s'agit d'un type valeur, alors le type sera encadré d'un nullable. Par exemple, si le type de retour était un `int`, on obtiendrait un `int?` lors de l'appel, qui serait égal à `null` si la variable était à `null`, ou qui aurait la valeur le cas contraire.

```
public class TestClass
{
    public int Valeur { get; set; }
    public string ValeurString { get; set; }
    public void Methode() { }
    public int MethodeInt()
    {
        return 42;
    }
    public string MethodeString()
    {
        return "valeur";
    }
}

TestClass c = null;
int? valeur = c?.Valeur;
string valeurStr = c?.ValeurString;
c?.Methode();
int? retour = c?.MethodeInt();
string retourStr = c?.MethodeString();
```

Chapitre 4-2
Le pattern Chain of Responsibility

1. Description

Le pattern `Chain of Responsibility` construit une chaîne d'objets telle que si un objet de la chaîne ne peut pas répondre à une requête, il puisse la transmettre à son successeur et ainsi de suite jusqu'à ce que l'un des objets de la chaîne y réponde.

2. Exemple

Nous nous plaçons dans le cadre de la vente de véhicules d'occasion. Lorsque le catalogue de ces véhicules est affiché, l'utilisateur peut demander une description de l'un des véhicules mis en vente. Si une telle description n'a pas été fournie, le système doit alors renvoyer la description associée au modèle de ce véhicule. Si à nouveau, cette description n'a pas été fournie, il convient de renvoyer la description associée à la marque du véhicule. Une description par défaut est renvoyée s'il n'y a pas non plus de description associée à la marque.

Ainsi, l'utilisateur reçoit la description la plus précise qui est disponible dans le système.

Le pattern `Chain of Responsibility` fournit une solution pour mettre en œuvre ce mécanisme. Celle-ci consiste à lier les objets entre eux du plus spécifique (le véhicule) au plus général (la marque) pour former la chaîne de responsabilité. La requête de description est transmise le long de cette chaîne jusqu'à ce qu'un objet puisse la traiter et renvoyer la description.

Le diagramme d'objets UML de la figure 4-2.1 illustre cette situation et montre les différentes chaînes de responsabilité (de la gauche vers la droite).

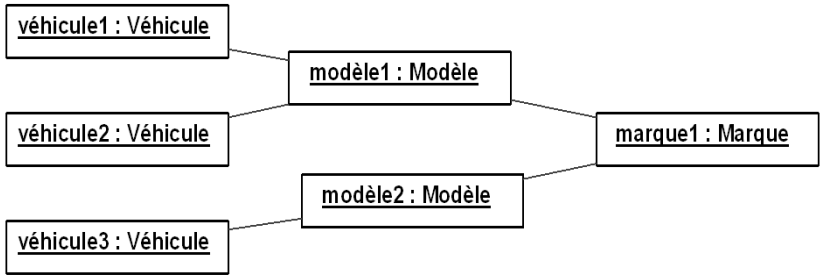


Figure 4-2.1 - Diagramme d'objets de véhicules, modèles et marques avec les liens de la chaîne de responsabilité

La figure 4-2.2 représente le diagramme des classes du pattern `Chain of Responsibility` appliqué à l'exemple. Les véhicules, modèles et marques sont décrits par des sous-classes concrètes de la classe `ObjetBase`. Cette classe abstraite introduit l'association `suivant` qui implante la chaîne de responsabilité. Elle introduit également trois méthodes :

– `getDescription` est une méthode abstraite. Elle est implantée dans les sous-classes concrètes. Cette implantation doit retourner soit la description si elle existe soit la valeur `null` dans le cas contraire.

– `descriptionParDéfaut` retourne une valeur de description par défaut, valable pour tous les véhicules du catalogue.

– `donneDescription` est la méthode publique destinée à l'utilisateur. Elle invoque la méthode `getDescription`. Si le résultat est `null`, alors s'il y a un objet `suivant`, sa méthode `donneDescription` est invoquée à son tour sinon c'est la méthode `descriptionParDéfaut` qui est utilisée.

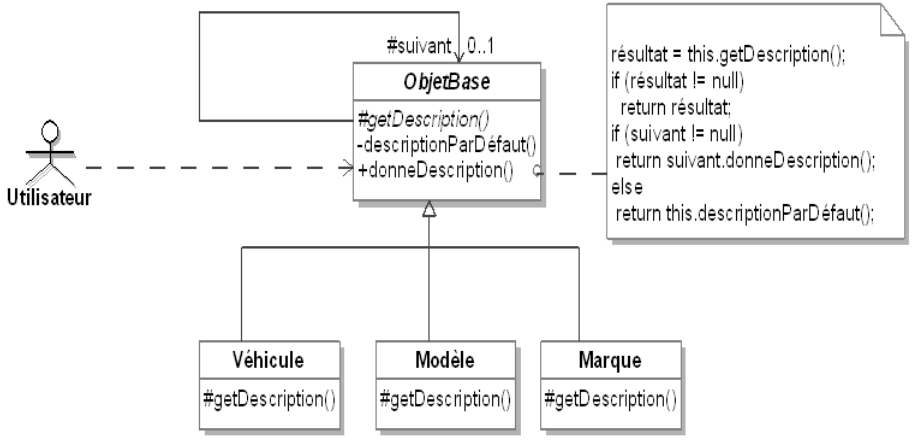


Figure 4-2.2 - Le pattern `Chain of Responsibility` *pour organiser la description de véhicules d'occasion*

La figure 4-2.3 montre un diagramme de séquence qui est un exemple de requête d'une description basée sur le diagramme d'objets de la figure 4-2.1.

Dans cet exemple, ni le `véhicule1`, ni le `modèle1` ne possèdent de description. Seule `marque1` possède une description qui est donc utilisée pour le `véhicule1`.

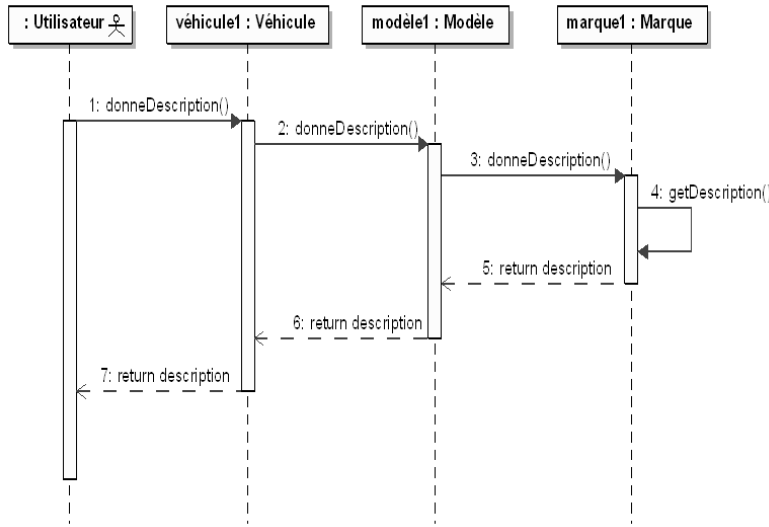


Figure 4-2.3 - Diagramme de séquence illustrant sur un exemple le pattern ***`Chain of Responsibility`***

3. Structure

3.1 Diagramme de classes

La figure 4-2.4 détaille la structure générique du pattern.

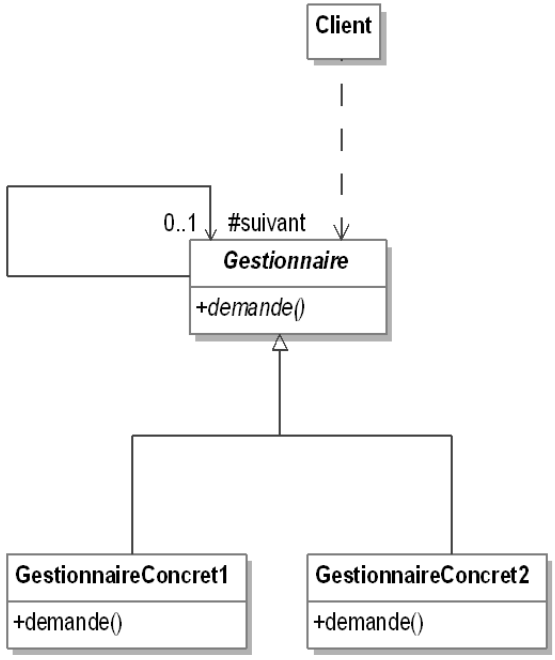


Figure 4-2.4 - Structure du pattern `Chain of Responsibility`

3.2 Participants

Les participants au pattern sont les suivants :

– `Gestionnaire` (`ObjetBase`) est une classe abstraite qui implante sous forme d'une association la chaîne de responsabilité ainsi que l'interface des requêtes.
– `GestionnaireConcret1` et `GestionnaireConcret2` (`Véhicule`, `Modèle` et `Marque`) sont les classes concrètes qui implantent le traitement des requêtes en utilisant la chaîne de responsabilité si elles ne peuvent pas les traiter.
– `Client` (`Utilisateur`) initie la requête initiale auprès d'un objet de l'une des classes `GestionnaireConcret1` ou `GestionnaireConcret2`.

3.3 Collaborations

Le client effectue la requête initiale auprès d'un gestionnaire. Cette requête est propagée le long de la chaîne de responsabilité jusqu'au moment où l'un des gestionnaires la traite.

4. Domaines d'application

Le pattern est utilisé dans les cas suivants :

– Une chaîne d'objets gère une requête selon un ordre qui est défini dynamiquement.
– La façon dont une chaîne d'objets gère une requête ne doit pas être connue de ses clients.

5. Exemple en C#

Nous introduisons maintenant l'exemple en langage C#. La classe `ObjetBase` est décrite à la suite. Elle implante la chaîne de responsabilité par la propriété `suivant`. Les autres méthodes correspondent aux spécifications introduites dans la figure 4-2.2.

```
using System;

public abstract class ObjetBase
{
 public ObjetBase suivant { protected get; set; }

 private string descriptionParDefaut()
 {
   return "description par défaut";
 }

 protected abstract string description { get; }

 public string donneDescription()
 {
   string resultat;
   resultat = this.description;
   if (resultat != null)
     return resultat;
   if (suivant != null)
     return suivant.donneDescription();
   else
     return this.descriptionParDefaut();
 }
}
```

Les trois sous-classes concrètes de la classe `ObjetBase` sont `Vehicule`, `Modele` et `Marque`, leur code source C# est présenté à la suite. La classe `Vehicule` gère une description simple fournie au moment de sa construction (le paramètre `null` est utilisé en cas d'absence de description).

```
using System;

public class Vehicule : ObjetBase
{
 protected string laDescription;
```

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.