THÉORIE : BLAZOR

Blazor est la technologie front-end dans la famille ASP.NET permettant de faire des SPAs. Le Program.cs se comporte exactement comme celui des APIs

Pour créer des composants, il faut créer des fichiers « .razor »
La balise @code{} permet de définir le code C# permettant de réagir aux évènements de l'interface graphique

Pour afficher une valeur qui est dans le code C# au niveau du HTML, utiliser la syntaxe @MaVariable ou @MonObjet.MaPropriete (ceci est valide aussi pour les instructions C# comme @if, @for, etc.) Si besoin : @(Une_instruction)

Pour récupérer un service dans un composant Blazor, il faut utiliser @inject :
@inject TypeDeService MonService

Pour ajouter du texte si besoin, utiliser la balise <text>@MaVariable</text>

THÉORIE : BLAZOR

Le cycle de vie d'un composant suit un ensemble de méthodes bien spécifiques

Dans la balise @code{}, il est possible de faire l'override des méthodes :

- OnInitialized[Async] ➔ initialisation du composant (une seule fois)
- OnAfterRender[Async] (bool) ➔ à chaque affichage du composant, le paramètre booléen indique s'il s'agit du premier rendu ou non

Attention : toute modification dans la méthode OnAfterRender ne sera PAS visible car le composant est déjà affiché. Si besoin de demander le rafraîchissement (à n'importe quel moment), utiliser la méthode StateHasChanged(). Soyez vigilants aux boucles de rendu

Les événements type @onclick peuvent être défini sur les composants HTML et invoquer une méthode C#

THÉORIE : BLAZOR

Une application Blazor s'exécute au sein du navigateur web et utilise de façon sous-jacente certains appels javascript natif (fetch)

Par défaut, les API en .NET utilise une sécurité à base de CORS permettant de filtrer et de valider d'où proviennent les appels, et quel type d'appel est autorisé

Il faut ajouter les services de gestion des CORS dans les services de l'API :
« services.AddCors() »

Et sur la variable app, avant d'appeler les méthodes MapPost etc., il faut appeler le middleware UseCors(c =>{}) et configurer la stratégie (c.AllowAnyMethods, etc.)

ÉTAPE 5 : INTERFACE GRAPHIQUE

L'interface graphique va se faire en Blazor WASM

Elle communiquera avec l'API afin de pouvoir démarrer le jeu mais aussi d'afficher la grille du joueur avec ses bateaux et la grille de l'adverse avec les tirs déjà effectués

Récupérer les ressources ici :

Faire une classe qui sera un singleton qui représente l'état actuel du jeu et stocke :

- char[,] est la grille du joueur
- bool?[,] est la grille de l'adversaire
  - Si NULL, jamais tiré
  - Si true => touché
  - Si false => raté

ÉTAPE 6 : MISE EN PLACE BLAZOR

Configurer le client http dans le Program.cs de l'application Blazor afin d'utiliser l'adresse de l'API comme adresse de base (définir la propriété BaseAddress)

Sur le composant principal, créer un bouton pour démarrer un nouveau jeu. Le composant principal fait un appel au démarrage de l'API et récupère les informations pour les stocker dans la classe d'état

Créer un composant Game qui récupère depuis les services le client HTTP ainsi que l'état de l'application contenant les deux grilles

Utiliser les deux tableaux dans l'état du jeu afin de pouvoir dessiner le contenu des grilles :

- Utiliser l'image « hit.png » pour les cases « touché »
- Utiliser l'image « miss.png » pour les cases « raté »
- Si grille du joueur : afficher la lettre du bateau si présent

ÉTAPE 6 bis : MISE EN PLACE BLAZOR

Dans l'interface graphique, si l'utilisateur clique sur une case de la grille adverse : jouer le coup au niveau de l'API

Traiter le retour de l'API :

1. Si le jeu est terminé et qu'il y a un gagnant, afficher « Vous avez gagné » ou « Vous avez perdu »
2. Mettre à jour la grille de l'adversaire avec l'icône correspondante
3. Si l'IA joue un coup de son côté, mettre à jour la grille du joueur avec le coup ainsi joué