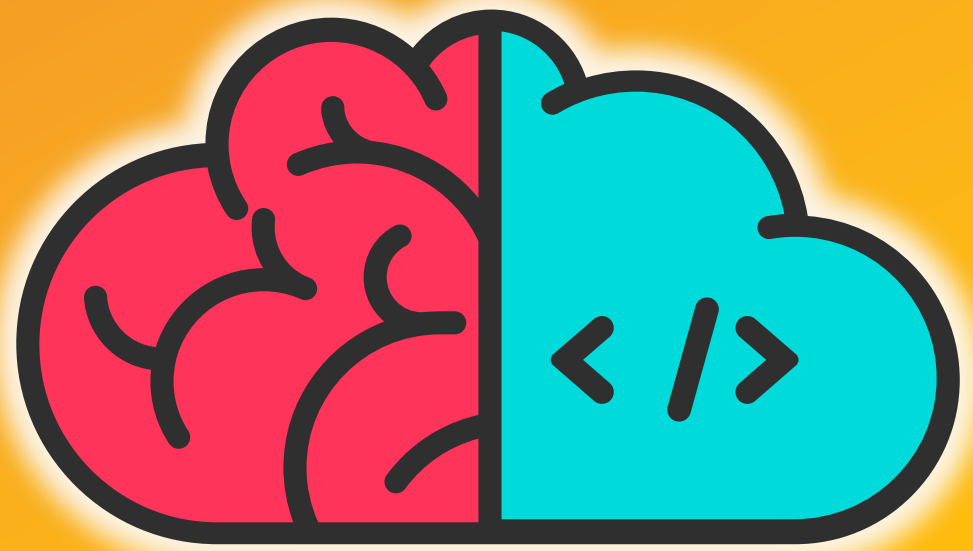
COURS C# / ASP.NET



Christophe MOMMER

https://www.hts-learning.com

PROJET

Pour apprendre ASP.NET, nous allons créer deux applications :

- Une API en ASP.NET Web API (Minimal API)
- Une application graphique (Blazor WebAssembly)

L'application sera le jeu de bataille navale

L'API sera chargée de gérer le jeu. Nous allons faire ensemble le développement de la partie solo contre l'ordinateur

Cet exercice servira de TP noté et sera fait en groupe de 2. L'évaluation tiendra compte de la qualité du code (ce que ChatGPT fait plutôt mal) et des fonctionnalités implémentées

Démonstration

THÉORIE : LES APIS EN .NET

ASP.NET propose deux modes de fonctionnement des APIs : MVC et Minimal APIs (controlerless)

Cette deuxième approche est recommandée pour la performance et la simplicité

Les méthodes MapGet/MapPost (et pour tous les autres verbes HTTP) doivent être appelées sur la variable app :

C#

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

app.MapGet("/", () => "Hello World!");

app.Run();
```

Les paramètres de la lambda peuvent être [FromBody] (POST), [FromServices] (injection de dépendances) ou encore [FromRoute] pour extraire une valeur de l'URL

THÉORIE : LES APIS EN .NET

La sérialisation des éléments se fait avec JSON. Il suffit de décrire un modèle (une classe) pour la lire, comme pour l'écrire :

```
app.MapGet("bonjour/{nom}/{prenom}",(
    [FromRoute] string nom,
    [FromRoute] string prenom) =>
    {
        return TypedResults.Ok(
            new Personne
            {
                Nom = nom,
                Prenom = prenom
            }
        );
    }
);
```

```
1 reference
class Personne
{
    1 reference
    public string Nom{ get; set; }
    1 reference
    public string Prenom { get; set; }
}
```

THÉORIE : LES APIS EN .NET

La désérialisation est automatique depuis le body d'une requête POST

```
app.MapPost("bonjour", (
    [FromBody] Personne personne) =>
    {
        return TypedResults.Ok(
            $"Bonjour {personne.Nom} {personne.Prenom}");
    }
);
```

THÉORIE : LES APIS EN .NET

L'injection de dépendances permet de configurer les services de l'application au niveau d'un point central pour que le container les fournisse à chaque élément le demandant (endpoint, classe, etc.)

Pour enregistrer un service, il faudra le faire sur builder.Services avec la méthode correspondant à sa durée de vie :

- Transient : éphémère
- Scoped : durée de vie limitée à un périmètre
- Singleton : instance unique partagée

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

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.)

THÉORIE : GRPC

GRPC permet d'échanger des données au format binaire entre les différentes applications

Souvent utilisé pour les échanges entre microservices car le contenu est concis et les échanges réduits (haute performance), il est possible de l'utiliser sur le web

Les échanges par le protocole gRPC nécessite de définir un contrat de communication (un fichier .proto) qui sera utilisé par le deux parties pour être compris

En ASP.NET, le tooling génère beaucoup d'éléments de façon automatique pour simplifier les échanges

Installer le package gRPC.AspNetCore sur le projet d'API pour activer le fonctionnement gRPC

ÉTAPE 1 : CRÉER LES PROJETS

Le projet se compose de 3 projets C# :

1. L'API
2. L'application Blazor
3. La librairie d'échange (les modèles)

Créer l'api avec la commande
« dotnet new webapi –minimal -n BattleShip.API »

Créer l'application Blazor WebAssembly
« dotnet new blazorwasm –e -n BattleShip.App »

Créer la librairie de modèles
« dotnet new classlib –n BattleShip.Models »

Créer la solution : « dotnet new sln –n BattleShip » et faire les liens (dotnet sln add [PROJET])

ÉTAPE 2 : CRÉATION DE GRILLE

Le jeu de bataille navale va être simplifié car c'est l'ordinateur qui va générer la position des bateaux (grille de 10x10)

On va utiliser un tableau à double dimension de char afin de déterminer les cases (char[,])

L'API doit se charger de générer deux grilles : une pour le joueur et une autre, gardée secrète, représentant le jeu de l'IA

Pour générer une grille :

- Un bateau aura une lettre attitrée (A => F) et une taille (1 => 4)
- Un bateau peut être placé en horizontal ou en vertical
- Il ne doit pas dépasser de la grille
- On obtient un aléatoire avec Random.Shared.Next(X) (X étant la valeur maximum - 1)
- La grille vide est composée uniquement de '\0', sinon elle contient la lettre du bateau

ÉTAPE 3 : API DE JEU

Préparons maintenant les endpoints afin de pouvoir exploiter notre jeu.
Au niveau de l'API, il sera nécessaire de proposer les opérations suivantes :

- Démarrer une nouvelle partie ➔ le retour de cet appel renverra les bateaux du joueur pour que l'application puisse l'afficher ainsi que l'identifiant de la partie qui a été créé

- Faire une attaque (passer l'identifiant du jeu ainsi que les coordonnées de l'attaque) ➔ le retour de cet appel renverra l'état du jeu (l'identité du gagnant si existant), l'état du tir (touché/raté) et l'attaque éventuelle de l'autre joueur (l'IA ici)

Pour que l'IA puisse répondre, il faudra prévoir une liste de coups possibles et les mélanger de façon aléatoire afin de les prendre dans l'ordre

ÉTAPE 4 : ALGORITHME DE JEU

Le joueur a tiré, il faut vérifier ce qu'il y a aux coordonnées indiquées. Si c'est '\0', cela signifie que la case est vide (raté). Si c'est une lettre, c'est qu'un bateau a été touché

Lorsque le tir a été effectué, vérifier si le jeu est terminé
Si oui, renvoyer comme quoi le jeu est terminé et que le joueur est gagnant
Si non, faire tirer l'IA et vérifier si le jeu est terminé. Si oui, l'IA est gagnante

Dans tous les cas, renvoyer toutes les informations à l'application graphique pour la mise à jour de l'interface

Pour vérifier si le jeu est terminé, la façon « rapide » est de parcourir les grilles et voir si elles contiennent encore une lettre de A à F

Il faut marquer les cases jouées dans la grille avec X (touché) ou O (raté). Il est possible de compter le nombre de X qui doit être égal à 13

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

ÉTAPE 7 : VALIDATION

Sur le web, toutes les données doivent être validées à un niveau serveur pour préserver la cohérence des données (anticorruption layer) et la sécurité

ASP.NET dispose d'un système de validation intégré mais qui montre vite ses limites

Au lieu de cela, installer le package FluentValidation (dotnet add package FluentValidation)

Pour créer un validateur, créer une classe qui hérite de AbstractValidator<Model>
Dans le constructeur, implémenter les règles avec RuleFor(x => …)

Enregistrer le validator en tant que IValidator<Model> dans le conteneur de services et l'importer dans les endpoints concernés pour valider le modèle avec la méthode Validate

Valider et faire le nécessaire pour le modèle « AttackRequest »

Note : si un endpoint a plusieurs codes de retour, utiliser Results<Code1, Code2, etc.>

ÉTAPE 8 : COMMUNICATION GRPC

Nous allons communiquer avec GRPC entre l'API et l'application Blazor (tout en gardant les endpoints REST pour le débug)

Installer l'extension vscode-proto3 pour avoir une coloration syntaxique et une aide à la complétion

Créer un dossier de solution pour stocker les fichiers proto
Créer un fichier proto pour les requêtes et les réponses

Exemple:

```
syntax = "proto3";

import "google/protobuf/wrappers.proto";

message AttackRequestGRPC {
    int32 col = 1;
    int32 row = 2;
}
```

```
service BattleshipService {
    rpc Attack(AttackRequestGRPC) returns AttackResponseGRPC;
```

ÉTAPE 8 bis : COMMUNICATION GRPC (API)

Lorsque tous les contrats ont été produits, modifier le fichier csproj de l'API pour inclure les contrats protobuf :

```
<ItemGroup>
  <Protobuf Include="..\requests_responses.proto" GrpcServices="Server" />
</ItemGroup>
```

Si le contrat a bien été créé et que les liens sont faits, il est possible de créer un service qui hérite du service automatiquement généré par le tooling

Le service automatiquement généré reprend le nom du service avec Base à la fin :

```
0 references
public class BattleshipGRPCService : BattleshipService.BattleshipServiceBase
{
```

Il faudra overrider les méthodes pour donner une implémentation
Attention : les collections en GRPC doivent être remplie après la création

ÉTAPE 8 ter : COMMUNICATION GRPC (API)

Il faut ensuite mapper notre nouveau service gRPC au niveau de ce qu'ASP.NET gère

Pour ceci, il suffira simplement d'appeler MapGrpcService avec le service que l'on souhaite mapper

Note : il est possible de tester le service gRPC sans implémenter le client grâce à une extension de Visual Studio Code (gRPC Clicker)

Attention: GRPC nécessite HTTP2,
il faut configurer l'application dans le appsettings.json

Avec cette configuration, le port 5001 tourne
en HTTP 1 &HTTP2

Et le port 5224 en HTTP1 pour les endpoints « historiques »

```
"Kestrel": {
  "Endpoints": {
    "Http2": {
      "Url": "http://localhost:5001",
      "Protocols": "Http1AndHttp2"
    },
    "Http1":{
      "Url": "http://localhost:5224",
      "Protocols": "Http1"
    }
  }
}
```

ÉTAPE 9 : COMMUNICATION GRPC (CLIENT)

Les appels GRPC depuis le navigateur ne sont pas encore possible en l'état. Heureusement, l'équipe ASP.NET met à disposition un package permettant aux applications web de réaliser les appels GRPC depuis Blazor WASM

➔ Installer le package Google.Protobuf, Grpc.Net.Client.Web, Grpc.Tools & Grpc.Net.Client sur le projet Blazor

1. Ajouter le support du fichier proto dans le csproj (en mode Client cette fois) + la classe qui override le client de base
2. Ajouter le code dans les services pour créer et utiliser un client compatible GRPC mais conserver le code des services REST

```
builder.Services.AddScoped(sp =>
{
    var httpClient = new HttpClient(new GrpcWebHandler(GrpcWebMode.GrpcWeb, new
    HttpClientHandler()));
    var channel = GrpcChannel.ForAddress("http://localhost:5001", new
    GrpcChannelOptions { HttpClient = httpClient });
    return new BattleshipService.BattleshipServiceClient(channel);
});
```

TP

Le TP consiste à améliorer le fonctionnement de l'applications réalisée

- Historiser les batailles (afficher les coups joués et pouvoir revenir en arrière)
- Ajouter un leaderboard et d'autres informations visuelle (« porte-avion coulé »)
- Ajouter un mode multi-joueur (entre deux joueurs humains) – **utiliser SignalR**
- Faire en sorte de pouvoir recommencer une partie
- Côté front : ajouter des images pour les bateaux plutôt que des lettres
- Permettre au joueur de placer ses bateaux à sa façon
- Ajouter de la sécurité sur l'application pour sécuriser les échanges et ajouter une gestion d'utilisateurs – **conseil : utiliser Auth0**
- Améliorer le comportement de l'IA (attaque randomisée par périmètre)
- Rajouter un niveau de difficulté (taille de la grille & intelligence IA)

Envoi des liens GitHub (n'oubliez pas les noms du groupe dans le README) à l'adresse **contact@hts-learning.com**