C# microservice

Introduction

L'objectif principal est de vous familiariser avec les concepts fondamentaux des microservices, en mettant particulièrement l'accent sur la mise en pratique de ces connaissances à travers le langage de programmation C#.

Les microservices représentent une évolution majeure dans la manière dont nous concevons et développons des applications. Au lieu de structures monolithiques, les microservices encouragent la décomposition d'une application en composants autonomes et interconnectés. Cette approche offre une agilité accrue, permettant aux équipes de travailler de manière indépendante sur des services spécifiques, tout en facilitant la maintenance, l'évolutivité et la résilience du système dans son ensemble.

Microservices

Les microservices sont un paradigme architectural qui consiste à construire une application comme une suite de petits services indépendants, déployables de manière autonome. Chaque microservice se concentre sur une fonctionnalité spécifique de l'application, communiquant avec d'autres microservices via des protocoles légers. Cette approche permet une flexibilité accrue, une évolutivité facile et une maintenance simplifiée. Chaque microservice peut être développé, déployé et mis à l'échelle indépendamment des autres, favorisant ainsi l'agilité et la résilience.

API Gateway

Une API Gateway agit comme une passerelle entre les clients et les microservices sous-jacents. Elle simplifie l'accès aux fonctionnalités du système en agrégeant les différentes API fournies par les microservices en un point d'entrée unifié. L'API Gateway gère la routage des requêtes, la sécurité, la validation, la transformation des données, et d'autres tâches liées à la gestion du trafic. Cela améliore la sécurité, la facilité d'utilisation et permet d'ajuster l'architecture sous-jacente sans impacter les clients.

Clients Légers

Un client léger, dans le contexte du développement logiciel, se réfère à une interface utilisateur minimale qui délègue la plupart des traitements au serveur. Contrairement aux clients lourds, les clients légers sont souvent des applications web ou des applications mobiles qui dépendent fortement des services back-end. Cette approche réduit la charge de travail côté client, permettant une expérience utilisateur plus fluide et des mises à jour centralisées. Les clients légers s'intègrent bien avec les architectures basées sur les microservices et les API Gateway, en tirant parti de leurs capacités d'accès simplifié aux fonctionnalités du système.

Avantage

En combinant les microservices pour une architecture agile et évolutive, une API Gateway pour simplifier l'accès aux services, et des clients légers pour offrir une expérience utilisateur optimale, les équipes de développement peuvent créer des systèmes robustes et flexibles. Cette approche permet de mieux répondre aux exigences changeantes, de faciliter la maintenance et d'améliorer la mise a l'échelle des applications modernes. En adoptant ces concepts, les développeurs peuvent construire des systèmes modulaires, interopérables et adaptables, répondant ainsi aux besoins croissants de l'environnement informatique contemporain.

Projet

Objectif

Le but de ce projet est de créer un gestionnaire de tâches avec un système de compte utilisateur.

Vous utiliserez les microservices, une API Gateway et le framework Blazor pour mettre en œuvre cette application.

Le front
Vous devez realiser une interface permetant de créer un compte puis de s'authentifier avec le compte.

Une fois connecté vous pourrez ajouter et supprimer des tâches de votre collection.

Vous pourrez également cocher une tache pour modifier son statut en terminé ou non.

API Gateway
Ce service a pour but d'exposer les différentes intéraction nécessaire au fonctionnement du front. Il sera également responsable de valider l'authentification de l'utilisateur (Jeton JWT).

Liste non exhaustive des fonctions:
- Recevoir un appel de création d'utilisateur
- Recevoir un appel de connexion
- Recevoir un appel d'ajout de tâche
- Recevoir un appel de suppression de tâche
- Recevoir un appel de validation de tâche

Micro service utilisateur
Ce service est responsable de tout ce qui est en lien avec la gestion d'un utilisateur.

Il utilisera une base sql server pour stocker les données des utilisateurs.

Ce micro service permettra de :
- Créer un compte
- Valider un combo login/pass

Un utilisateur possède a minima:
- un id
- un identifiant (mail, pseudo, nom)
- un mot de passe (on ne le stocke pas en clair en base ni en MD5)

Micro service tâches
Ce service est responsable de tout ce qui est en lien avec la gestion d'une tâche.

Ce micro service permettra de :
- Créer une tâche
- Supprimer une tâche
- Valider/Unvalider une tâche

Créer le projet

Création de la solution + front
Vous aurez besoin d'ouvrir Visual Studio 2022 et de créer un nouveau projet. Créer un projet avec comme premier composant ‹ Blazor Web App ›.

Utilisez ces paramètres

```
Framework .Net 8
Authentification : Aucun
Décocher HTTPS
Mode de rendu intéractif : Serveur
```

Vous pouvez créer le projet.

Ajout du premier micro service

Dans l'explorateur de solution (menu a droite par défaut) fait un clic droit sur la solution et ajouter un nouveau projet.

Cette fois-ci ajoutez un projet ‹ API web ASP.NET Core ›

Comme précedemment voici les paramètres à utiliser:

```
Framework .Net 8
Authentification : Aucun
Décocher HTTPS
```

API Gateway / Second micro service

Répétez la même opération que pour le premier micro service. La gateway et le second micro service seront eux aussi des API web.

Micro service utilisateur

Nous allons mettre en place ce premier micro service qui va permettre de créer et retrouver des utilisateurs dans une base de donnée.

Créer le controlleur utilisateur

Clic droit sur le dossier Controller puis Ajouter -> Classe. Dans le menu Web on sélectionne ‹ Controleur d'API avec actions ›

Nous voici avec un squelette de controlleur qui ne fait pas grand chose, on y reviendra après.

Créer un classe Utilisateur

Nous allons créer une classe Utilisateur qui nous permettra de les manipuler plus facilement.

Commencez par créer un dossier Entities dans lequel vous ajouterez une classe Utilisateur.

Vous êtes libre de mettre les champs qui vous semble utile dans cette classe. Cependant il vous faudra a minima, un champ id, un champ identifiant et un champ motDePasse.

voici un exemple

```
using System.ComponentModel.DataAnnotations;

namespace utilisateur.Entities
{
    public class Utilisateur
    {
        public int Id { get; set; }
        public string Prenom { get; set; }
        public string Nom { get; set; }
        public string Email { get; set; }
        public string Pass { get; private set; }
        public string NomComplet => Nom + " " + Prenom;

        // Constructeur
       public Utilisateur(int id, string nom, string prenom, string email, string pass)
        {
            Id = id;
            Prenom = prenom;
            Nom = nom;
```

```
            Email = email;
            Pass = pass;
        }

        public void changePass()
        {
            this.Pass = "muchSecure";
        }

        public bool isPassSecure()
        {
            if(Pass.Length > 6)
            {
                return true;
            }
            return false;
        }
    }
}
```

Vous remarquerez peux être la propriété NomComplet qui est une propriété calculé. La propriété pass a aussi la spécificité d'avoir son setter en privée.

Plus d'info ici : https://learn.microsoft.com/en-us/dotnet/csharp/properties

Ajout de la base de donnée

Pour pouvoir sauvegarder nos données, nous allons utiliser l'entity framework. C'est un framework qui s'occupe d'abstraire les requêtes vers la base pour nous.

Pour cela, il faut l'ajouter à notre projet utilisateur. Noue en profitons pour ajouter dotnet-ef qui permettra d'initialiser notre base de donnée

Il faut executer ces commandes dans le répértoire du projet utilisateur depuis un terminal.

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet tool install --global dotnet-ef --version 8.*
```

Un fois cela fait, on peux ajouter notre classe qui permettra de se connecter a cette base.

On commence par créer un dossier Data dans lequel on ajoute une classe DataContext.cs

Voici le corps de cette classe

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using System.Collections.Generic;
using utilisateur.Models;

namespace utilisateur.Data
{
    public class DataContext : DbContext
    {
        protected readonly IConfiguration Configuration;

        public DataContext(IConfiguration configuration)
        {
            Configuration = configuration;
```

```
        }

        protected override void OnConfiguring(DbContextOptionsBuilder options)
        {
                                            //    Connexion    a    la    base    sqlite
options.UseSqlite(Configuration.GetConnectionString("WebApiDatabase"));
        }

        // On déclare que notre base aura une table Utilisateurs qui contiendra des
objets de type Utilisateur
        public DbSet<Utilisateur> Utilisateurs { get; set; }


    }
}
```

Une fois ceci fait il faut ajouter dans le fichier appsettings.json les lignes suivantes pour que notre programme sache dans quel fichier stocker la base

```
"ConnectionStrings": {
  "WebApiDatabase": "Data Source=Utilisateur.db"
},
```

Enfin dans le Program.cs il faut dire a notre service qu'il peux utiliser cette base la.

```
...
builder.Services.AddDbContext<DataContext>();

var app = builder.Build();
```

Nous pouvons enfin initialiser notre fichier avec les bonnes tables. Pour cela on reprend le terminal et on va créer une migration puis l'appliquer.

```
# On crée une migration que l'on nomme Init
dotnet ef migrations add Init
# On l'applique
dotnet ef database update
```

Utiliser la base dans le controller

Dans le controlleur Utilisateur ajoutez une propriété privée en lecture seule de type DataContext.

Ensuite, ajoutez le en paramètre du constructeur pour l'initialiser.

Vous devriez avoir ceci.

```
private readonly DataContext _context;

public UtilisateurController(DataContext context)
{
    _context = context;
}
```

Le controlleur est instancié automatiquement par notre application au démarrage. Le contexte sera automatiquement crée et injecté dans le constructeur par injection de dépendance.

Doc : https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection

Récuperer tout les utilisateurs

Pour accéder a notre base on utilise `_context.Utilisateurs`. On peux récuperer l'ensemble des éléments avec la fonction `ToListAsync()`.

Ci dessous un exemple de fonction qui permet de renvoyer l'ensemble des utilisateurs de la base.

```
// GET: api/user/Utilisateur
[HttpGet]
public async Task<ActionResult<IEnumerable<Utilisateur>>> GetUtilisateurs()
{
    return Ok(await _context.Utilisateurs.ToListAsync());
}
```

Comme nous utilisons des fonctions asynchrone il faut utiliser le mot clé `await` pour attendre l'éxécution de l'opération et ajouter le mot clé `async` avant le type de retour.

Si on examine le type de retour `Task<ActionResult<IEnumerable<Utilisateur>>>`

Nous avons besoin d'un type de retour qui permet l'asynchrone, ce type est `Task`.

Une réponse web comporte également un status de retour (200 OK, 404 NOT FOUND, etc), c'est le type ActionResult qui symbolyse cela.

Nous allons renvoyer une list d'Utilisateur, l'interface IEnumerable indique que nous renvoyont un élément itérable, cet élément sera composé d'Utilisateur. `toListAsync()` renvoi une `List` qui implémente IEnumerable, nous aurions très bien pu mettre `List` a la place

Voici quelques exemples de code de retour

```
return Ok(value); // 200
return CreatedAtAction(nameof(Getter), new { id = value.Id }, value); // 201
return NoContent(); // 204
return BadRequest(); // 400
return NotFound(); // 404
```

Récuperer un utilisateur spécifique

Voici le corps de la fonction `GetUtilisateur` qui permet de retourner un utilisateur par son id.

Implementez cette fonction

```
// GET: api/user/Utilisateur/5
[HttpGet("{id}")]
public async Task<ActionResult<Utilisateur>> GetUtilisateur(int id)
{
  // On récupère l'utilisateur via le _context

  // On verifie si il est null si oui on renvoi NotFound

  // On renvoi l'utilisateur
}
```

Insertion en base

Nous avons la possibilité de renvoyer des utilisateurs mais pas d'en insérer, remédions a ce problème.

```
// POST: api/user/Utilisateur
[HttpPost]
public async Task<ActionResult<Utilisateur>> PostUtilisateur(Utilisateur utilisateur)
{
    // On ajoute notre utilisateur dans la base
    _context.Utilisateurs.Add(utilisateur);
    // On sauvegarde la modification
    await _context.SaveChangesAsync();

   // On retourne l'utilisateur nouvellement crée en appelant la fonction CreatedAtAction
```

```
        return CreatedAtAction(nameof(GetUtilisateur), new { id = utilisateur.Id },
utilisateur);
}
```

Login

Implémentez un fonction qui sera accessible en POST sur l'URL ‹ /login › du controller.

Elle prendra en paramètre l'identifiant de l'utilisateur ainsi que le pass.

Je vous conseille d'utiliser `FirstOrDefaultAsync` et de regarder comment on fait une query LINQ en mode lambda.

```
// spoiler
pokemons.Where(p => p.Name == "Dracofeu");
```

Aller plus loin

Avant la prochaine séance pour pouvez commencer a regarder l'application front en Blazor et remplacer la génération de la page Weather par un appel HTTP a un micro service. L'API weather est deja implémenté quand vous créez un projet API avec les exemples inclu.

Il faut dans un premier temps ajouter le HTTPClient dans le Program.cs. Puis l'injecter dans la page ou vous souhaitez l'utiliser. Enfin, au chargement de la page, envoyer la requête pour récupérer les données météo.

```
// Un exemple de requete HTTP JSON
pokemons = await HttpClient.GetFromJsonAsync<PokemonClass[]>("http://localhost:5008/
ListeDePokemon");
```

Fixer le port d'execution du serveur web

Par défaut un port aléatoire est généré à la création d'un projet, vous pouvez le changer en allant modifier `Properties/launchSettings.json`.

Voici un exemple avec le port 5002

```
"applicationUrl": "http://localhost:5002",
```