

## Séance 10 : Initiation à l'ASP.NET – Partie IV

☑ Introduction aux *microservices*

---

INF27507 – Technologies du commerce électronique

Prof. Yacine YADDADEN, Ph. D.

# Plan

1. Introduction et Contexte
2. Les deux différents types d'architectures
  - a. Qu'est-ce que l'architecture *monolithique* ?
  - b. Qu'est-ce qu'une architecture en *microservices* ?
  - c. Bilan → avantages et inconvénients
3. Les *microservices* et les *conteneurs*
4. Les outils et configurations nécessaires
  - a. Ocelot
  - b. Microsoft Entity Framework Core
  - c. SWAGGER
4. Mise en place d'une architecture en *microservices* :
  - a. Création de la solution et de la passerelle
  - b. Installation des paquets nécessaires
  - c. Configuration de la passerelle
  - d. Création des autres services
  - e. Configuration des bases de données
  - f. Documentation de l'architecture
  - g. Interaction entre la passerelle et les services
  - h. Interaction entre les différents services
5. Questions et discussion

# Introduction et Contexte

- **Contexte :**

- Les applications développées deviennent de plus en plus complexe,
- Les équipes qui interviennent sont de plus en plus nombreuses,
- Les besoin des entreprises deviennent de plus en plus complexes.

- **Problématique :**

- Difficulté à maintenir l'application sachant qu'il y a plusieurs technologies,
- Difficulté à faire évoluer l'architecture courante en y ajoutant de nouvelles fonctionnalités.

- **Solution :**

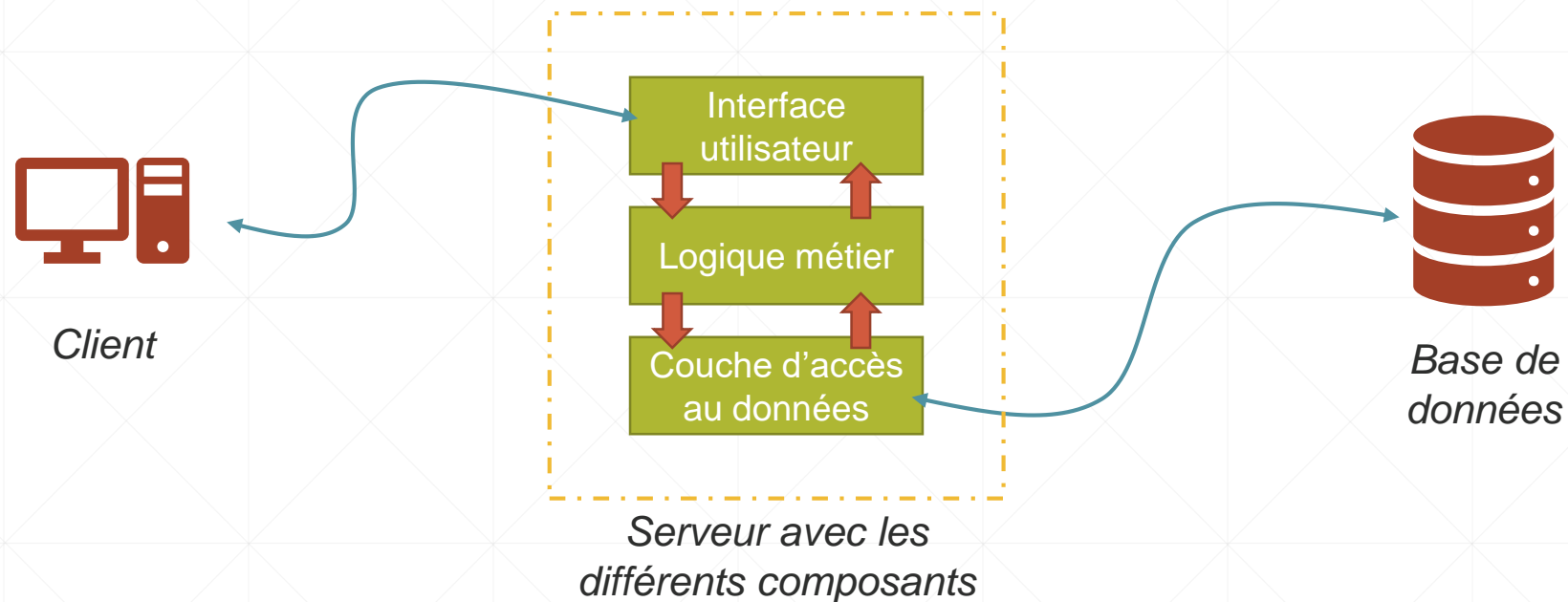
- Utilisation d'une architecture en *microservices*,
- C'est une alternative à l'architecture *monolithique* classique remédiant à ses défauts.

# Architecture *monolithique*

- **Qu'est-ce qu'une architecture monolithique ?**
  - *C'est un style d'architecture logiciel où l'ensemble des composants de l'application sont construits autour d'une seule et unique entité. Le résultat est une seule et même application.*
- Généralement, on fera appel à *motif d'architecture logicielle* afin de faciliter le développement, comme :
  - MVP (*Model-View-Presentation*),
  - **MVC** (*Model-View-Controller*),
  - MVVM (*Model-View-ViewModel*), ...
- Elle aura, généralement, qu'une seule base de données pour la *persistance*,
- C'est l'architecture la plus commune et celle qui a été la plus utilisée.



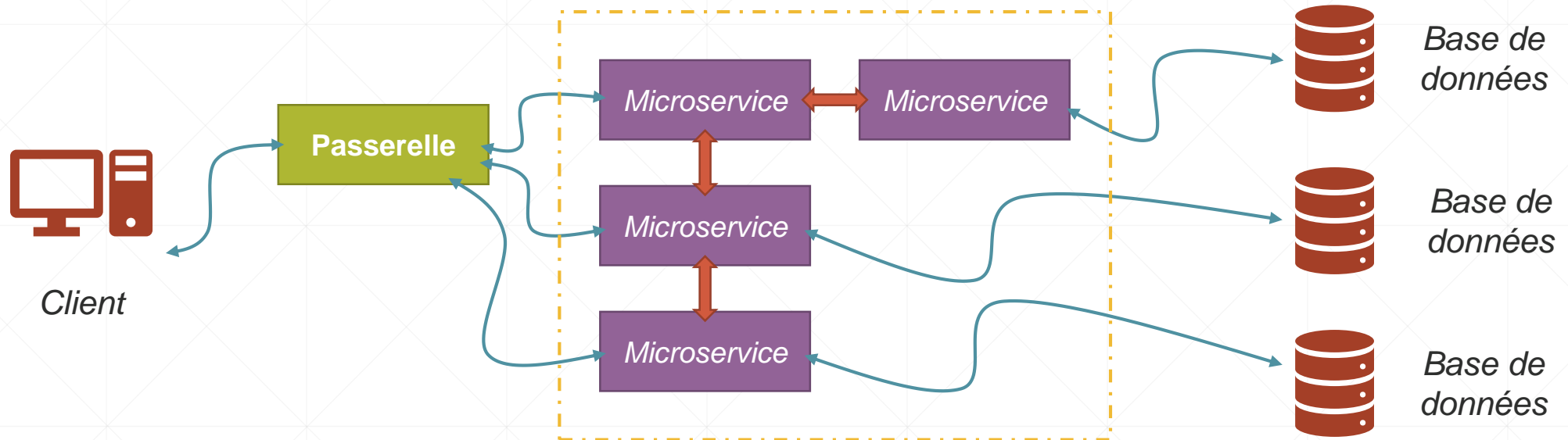
## Aperçu → Architecture *monolithique*



# Architecture en *microservices*

- **Qu'est-ce qu'une architecture en microservices ?**
  - *C'est un style d'architecture logiciel où l'application développée est construite autour de plusieurs microservices indépendants interagissant et communicants entre eux. Chaque microservice s'occupe de l'exécution d'une tâche particulière.*
- Généralement, cette architecture est déployées sous forme d'**API REST**.
- Cette architecture n'est pas forcément déployée sur un seul et même serveur, mais distribuée sur plusieurs serveurs,
- La *communication* entre les microservices s'effectuer à travers de **requêtes HTTP**,
- Pour la *persistance* des données, plusieurs bases de données sont utilisées,
- Elle a été discuté pour la première fois par **Adrian Cockcroft** (*ingénieur en chef chez Netflix*) en 2011.

# Aperçu → Architecture en *microservices*



# Architecture monolithique vs. *microservices*

## Avantages

- ✓ Le développement se fait rapidement,
- ✓ Gestion des données est plus simplifiée,
- ✓ Gestion et maintenance plus aisée :
  - Cas d'une petite application.

## Inconvénients

- ✗ Difficulté de mise à l'échelle,
- ✗ L'application doit être redéployée :
  - Même en cas de modification mineure.
- ✗ Difficulté du développement, car tous les développeurs de l'équipe travaillent sur une même application.



# Architecture *monolithique* vs. microservices

## Avantages

- ✓ Évolutivité et flexibilité,
- ✓ Facilité de déploiement :
  - Chaque *microservice* est indépendant.
- ✓ Réduction de la complexité :
  - Chaque *microservice* effectue une tâche.

## Inconvénients

- ✗ Gestion de la complexité :
  - Car les *microservices* sont interconnectés
- ✗ Coût élevé du développement :
  - Plusieurs *développeurs* (équipes).
- ✗ Gestion des données plus complexe :
  - Plusieurs bases de données sont utilisées.

**IMPORTANT :** *le choix de l'architecture adéquate dépend de plusieurs paramètres.*

# Architecture en *microservices*

## → *Utilisation des conteneurs – Docker & Kubernetes*

Afin de faciliter le déploiement des différents *microservices*, on fera appel à :

- Des *conteneurs*

- ✓ **Définition** : « *c'est environnement d'exécution léger et portable permettant d'isoler une application et ses dépendances logicielle du reste du système d'exploitation.* ».

- ✓ La technologie utilisée est Docker permettant une virtualisation au niveau du système d'exploitation. Ils sont *léger* et *facile* à *déployer*.

- Un *gestionnaire* de conteneurs

- ✓ **Définition** : « *c'est un système Open Source permettant la gestion et l'orchestration des conteneurs. Il permet de déployer, gérer et mettre à l'échelle des applications conteneurisées.* ».

- ✓ La technologie utilisée est Kubernetes, elle a été développée par **Google**.



# Architecture en *microservices*

## → *Utilisation des conteneurs – Docker*

Afin de faciliter le déploiement des différents *microservices*, on fera appel à :

- Une plateforme de *déploiement*

- ☑ **Définition** : « *c'est une plateforme de cloud computing permettant de déployer, gérer et mettre à l'échelle des applications conteneurisées basées sur l'utilisation de microservices.* ».

- ☑ La technologie utilisée est [Azure](#) qui contient différents services en lien avec les conteneurs :

- **Azure Kubernetes Service** (AKS) : service de gestion d'orchestration de conteneurs *Kubernetes*.

- **Azure Container Instances** (ACI) : service qui permet de lancer rapidement et facilement des conteneurs sans avoir à gérer une infrastructure de cluster *Kubernetes*.

- **Azure Container Registry** (ACR) : registre privé de conteneurs qui permet de stocker, gérer et distribuer des images de conteneurs Docker et d'autres formats de conteneurs.

- **Azure Service Fabric** : plateforme de développement d'applications de microservices qui fournit un modèle de programmation et une infrastructure de gestion pour les applications distribuées.

- ☑ Elle a été déployée et maintenue par **Microsoft**.



# Les outils et configurations nécessaires

## → *Ocelot*

- **Définition :** « *C'est une bibliothèque Open Source pour .NET permettant la création d'une passerelle d'API ou API Gateway.* »,
- Son objectif est d'*agréger* plusieurs microservices en un seul point de terminaison d'API,
- Il permet de *faciliter la communication* avec le client à travers des *requêtes* HTTP,
- De plus, il offre différentes fonctionnalités telle que la répartition de charge.
- Afin de pouvoir l'installer, il faut utiliser la commande suivante :
  - `Install-Package Ocelot`

# Les outils et configurations nécessaires

## → *Microsoft Entity Framework Core*

- **Définition :** « *C'est une Framework Open Source pour .NET permettant l'utilisation d'un **ORM**.* »,
- Ce dernier permet le passage d'une représentation *relationnelle* à une représentation *objet* et vice versa,
- Il fonctionne avec différents serveurs de gestion de bases de données tels que :
  - Microsoft SQL Server, MySQL, SQLite, ...
- Afin de pouvoir l'installer, il faut utiliser la commande suivante :
  - `Install-Package Microsoft.EntityFrameworkCore.Tools`
  - `Install-Package Microsoft.EntityFrameworkCore.Design`
  - `Install-Package Microsoft.EntityFrameworkCore.SqlServer`



# Les outils et configurations nécessaires

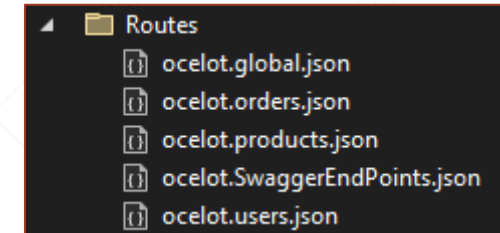
## → SWAGGER

- **Définition :** « *C'est un outil Open Source permettant la conception, la documentation et le test des API REST.* »,
- Il permet de générer automatiquement une documentation détaillée,
- Il se base sur l'utilisation de la spécification **OpenAPI** qui permet de décrire les opérations d'une API, les paramètres d'entrée et de sortie, schéma des données, ...
- Afin de pouvoir l'installer, il faut utiliser la commande suivante :
  - `Install-Package Swashbuckle.AspNetCore`
- Afin de pouvoir l'intégrer à **Ocelot** et d'avoir une interface unifiée, il faut :
  - `Install-Package MMLib.SwaggerForOcelot`

# Mise en place d'une architecture en *microservices*

## → 1. Création de la solution et de la passerelle

- Pour cela, il faut suivre les étapes suivantes :
  1. ouvrir **Microsoft Visual Studio** et créer une nouvelle solution → EC\_MicroServices,
  2. Le projet initial sera de type **Web API** en **ASP.NET** → EC\_GateWay,
  3. Il faut créer un dossier qui contiendra les différentes routes → routes.
- Chacune des routes sera définie à l'aide d'un fichier de configuration :
  - ocelot.*microservive\_name*.json
- De plus, on aura un fichier de configuration .json *global* :
  - ocelot.*microservive\_name*.json
- De plus, on aura un fichier de configuration .json pour SWAGGER :
  - ocelot.*SwaggerEndpoints*.json




# Mise en place d'une architecture en *microservices*

## → 1. Création de la solution et de la passerelle

- Il faut configurer la solution pour le *lancement* des différents projets *en même temps* :
  - Voir dans **Propriétés** de la solution, puis dans **Projet de démarrage**,
  - Choisir **Plusieurs projets de démarrage** et sélectionner l'ensemble des projets.
- Pour la *passerelle*, il faut :
  - Se rendre dans **Propriétés** du projet, puis **Déboguer, Général**,
  - Dans **Profils de lancement de débogage**, il faut :
    - Cocher la case pour **Lancer le navigateur**
    - URL : *swagger*
    - URL de l'application : <http://localhost:5000>

*Définir le bon port suivant  
l'architecture définie (diapo 19)*



# Mise en place d'une architecture en *microservices*

## → 2. Installation des paquets nécessaires

Ensuite, il est nécessaire d'installer différents paquets à savoir :

- Au niveau de la *passerelle* ou EC\_GateWay, il faut installer :
  - **Ocelot**
  - **MMLib.SwaggerForOcelot**
- Pour le reste des *services*, il faut installer :
  - **Microsoft.EntityFrameworkCore.Tools**
  - **Microsoft.EntityFrameworkCore.Design**
  - **Microsoft.EntityFrameworkCore.SqlServer**
  - **Swashbuckle.AspNetCore**

# Mise en place d'une architecture en *microservices*

## → 3. Configuration de la passerelle avec SWAGGER

Il faut se rendre au niveau du fichier `Program.cs` dans le projet `EC_Gateway` :

- Déclarer le dossier *routes* :

```
var routes = "Routes";  
builder.Configuration.AddOcelotWithSwaggerSupport(options =>  
{  
    options.Folder = routes;  
});
```

- Déclarer les services :

```
builder.Services.AddOcelot(builder.Configuration);  
builder.Services.AddSwaggerForOcelot(builder.Configuration);
```

- Utiliser les services :

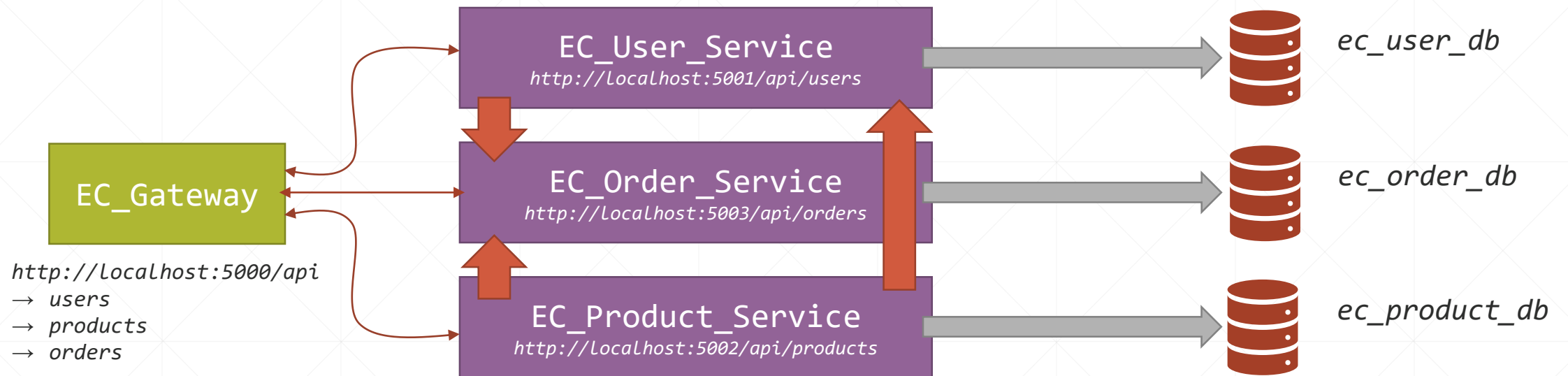
```
app.UseSwaggerForOcelotUI(options =>  
{  
    options.PathToSwaggerGenerator = "/swagger/docs";  
});  
app.UseOcelot().Wait();
```



# Mise en place d'une architecture en *microservices*

## → 4. Création des autres services

Dans le cadre de l'exemple, il faudra créer les *microservices* suivants :

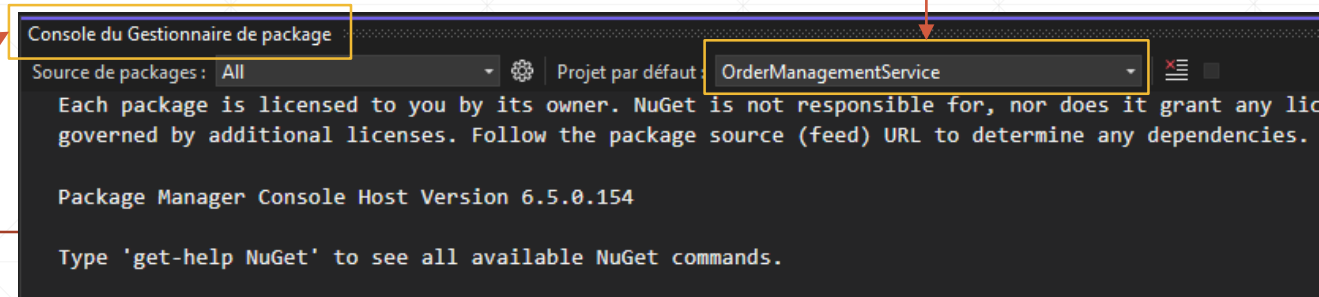


# Mise en place d'une architecture en *microservices*

## → 5. Configuration des bases de données

- Comme il y a trois différents *microservices*, il est nécessaire de créer trois bases de données différentes :
  - Service pour les *utilisateurs* → *ec\_user\_db*
  - Service pour les *produits* → *ec\_product\_db*
  - Service pour les *commandes* → *ec\_order\_db*
- Il est nécessaire de créer trois différents fichiers → *Service\_NameDbContext.cs*
- Il faut s'assurer d'exécuter les deux commandes nécessaires pour la génération des fichiers de migration ainsi que leur application.

Se rendre dans l'outil en ligne de commandes



# Mise en place d'une architecture en *microservices*

## → 6. Documentation de l'architecture

Pour documenter l'architecture en *microservice* mise en place, il faut :

- Configurer **SWAGGER** au niveau de chaque *projet (microservice)* → Program.cs
- Déclarer le *microservice* dans le fichier Routes/ocelot.SwaggerEndpoints.json

La valeur de Key doit être la même  
l'attribut **SwaggerKey** dans le fichier  
Routes/ocelot.users.json

```
{
  "SwaggerEndpoints": [
    {
      "Key": "users",
      "TransformByOcelotConfig": true,
      "Config": [
        {
          "Name": "Users Service (API REST)",
          "Version": "1.0",
          "Url": "http://localhost:5001/swagger/v1/swagger.json"
        }
      ]
    }
  ]
}
```

# Mise en place d'une architecture en *microservices*

## → 7. Interaction entre la passerelle et les services

Il faut modifier les fichiers suivants :

- Le fichier de configuration dans **Routes/ocelot.global.json** :

```
{
  "GlobalConfiguration": {
    "BaseUrl": "http://localhost:5000"
  }
}
```

Configuration de l'URL de base  
pour l'accès à la passerelle

- Les fichiers de configuration **Routes/ocelot.service\_name.json** :

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/users/{everything}",
      "DownstreamScheme": "http",
      "SwaggerKey": "users",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 5001
        }
      ],
      "UpstreamPathTemplate": "/api/users/{everything}",
      "UpstreamHttpMethod": [
        "GET",
        "POST",
        "DELETE"
      ]
    }
  ]
}
```

Informations sur le service concerné vers où sera transmise la requête

Informations sur le point d'entrée à travers la passerelle et les méthodes autorisées

# Mise en place d'une architecture en *microservices*

## → 8. Interaction entre les différents services

- Comme les différents *microservices* sont indépendants, il y a certaines contraintes :
  - ✗ Un *microservice* ne peut interagir directement avec la base de données d'un autre *microservice*,
- Pour y remédier, il faut faire en sorte :
  - ✓ D'établir la communication à travers des requêtes **HTTP**,
  - ✓ Il faut utiliser la bibliothèque : `using System.Net;`
  - ✓ Il utilise le code suivant :

```
private HttpClient _httpClient;  
private JsonSerializerOptions _options;  
  
public ProductController()  
{  
    _httpClient = new HttpClient();  
    _options = new JsonSerializerOptions  
    {  
        PropertyNamingPolicy = JsonNamingPolicy.CamelCase  
    };  
}
```



# Mise en place d'une architecture en *microservices*

## → 8. Interaction entre les différents services

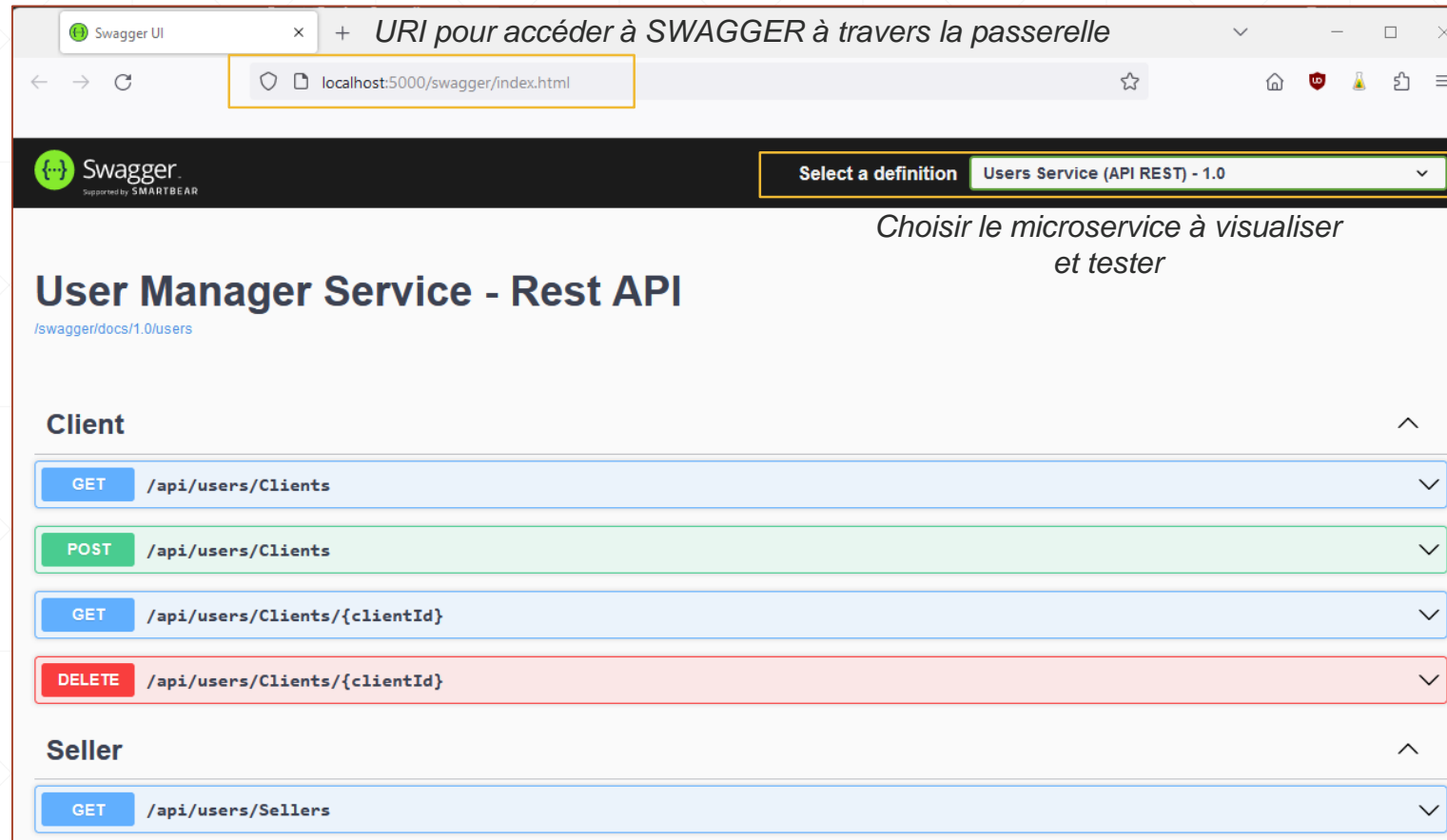
- Pour y remédier, il faut faire en sorte :

☑ Il utiliser le code suivant :

```
public async Task<IActionResult> AddProduct([FromBody] Models.Product model)
{
    Il faut que ça soit une requête asynchrone
    try
    {
        HttpResponseMessage response = await _httpClient.GetAsync($"http://localhost:5000/api/users/sellers/{model.SellerId}");
        La méthode à utiliser (GET) L'URI de la ressource à récupérer
        string responseContent = await response.Content.ReadAsStringAsync();
        Models.Seller? seller = JsonSerializer.Deserialize<Models.Seller>(responseContent, options);
        Opération de désérialisation format JSON en objet Seller
        if (response.IsSuccessStatusCode) { }
    }
    Dans le cas où le résultat de la requête renvoie un code 2xx
    catch (Exception) {}
}
```

# Mise en place d'une architecture en *microservices*

## → Aperçu du résultat



# Questions & Discussion

---

# Bibliographie

1. Gammelgaard, C. H. (2021). *Microservices in. NET*. Simon and Schuster.
2. Hoffman, K. (2017). Building microservices with ASP. NET Core: develop, test, and deploy cross-platform services in the cloud.
3. Baptista, G., & Abbruzzese, F. (2019). *Hands-On Software Architecture with C# 8 and. NET Core 3: Architecting software solutions using microservices, DevOps, and design patterns for Azure Cloud*. Packt Publishing Ltd.