

CHAPITRE 6 :

Les Microservices avec ASP.NET Core

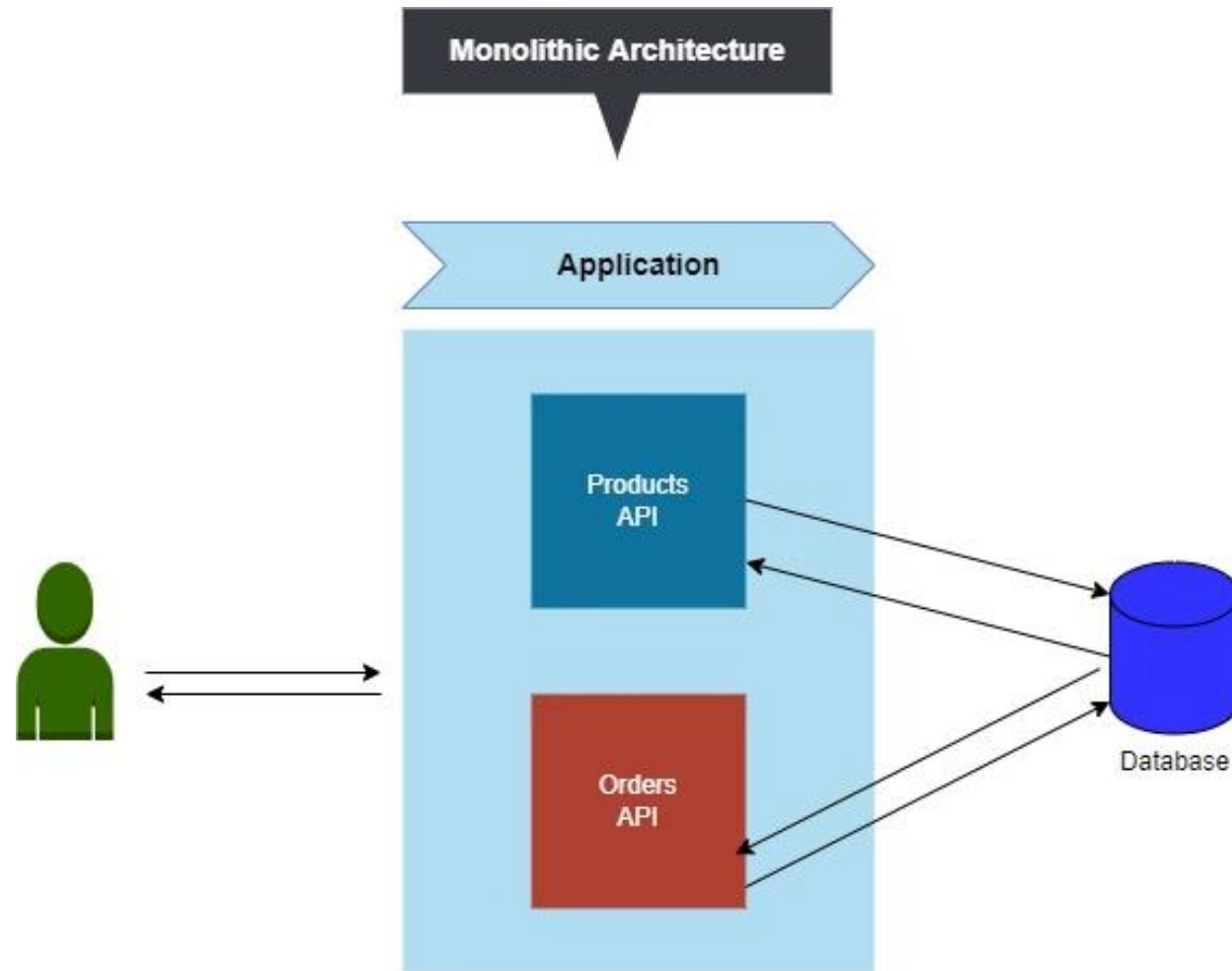
Que sont les Micro services ?

- Le concept de microservices est l'approche architecturale pour créer des applications de petite à grande échelle.
- Avec cette approche architecturale, une application est décomposée en composants plus petits, indépendants les uns des autres.
- Contrairement à l'architecture monolithique, où toutes les fonctionnalités sont ciblées pour être intégrées dans un seul projet/application.
- Les microservices permettent de séparer les fonctionnalités pour se développer de manière plus modulaire et tous les modules fonctionnent ensemble pour accomplir les tâches spécifiques ciblées.

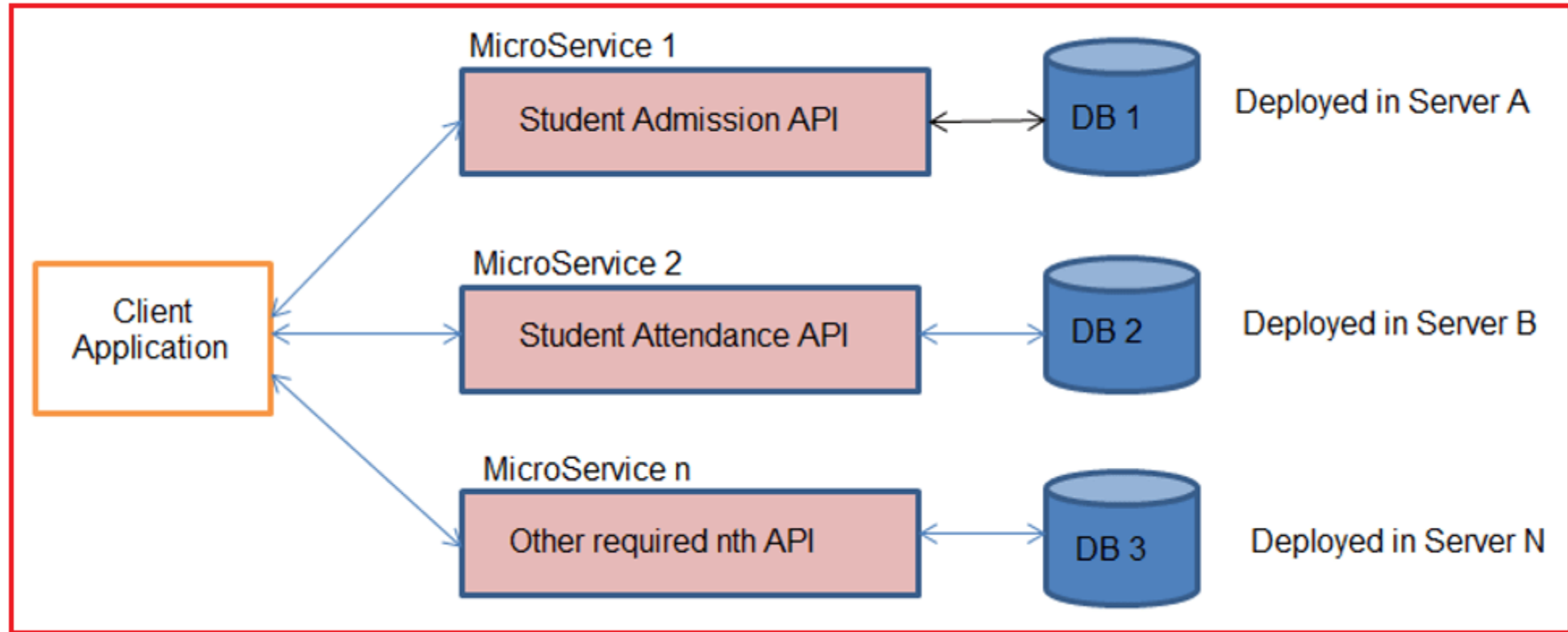
Avantages des microservices

- Les microservices offrent aux équipes de développement et aux testeurs une approche plus rapide grâce au développement distribué.
- Haute évolutivité.
- Résilience & Indépendance : si un service n'a pas fonctionné, l'ensemble de l'application ne tombera pas, contrairement au modèle d'application monolithique.
- Déploiement facile.
- Accessibilité pour le développement : modification plus facile.

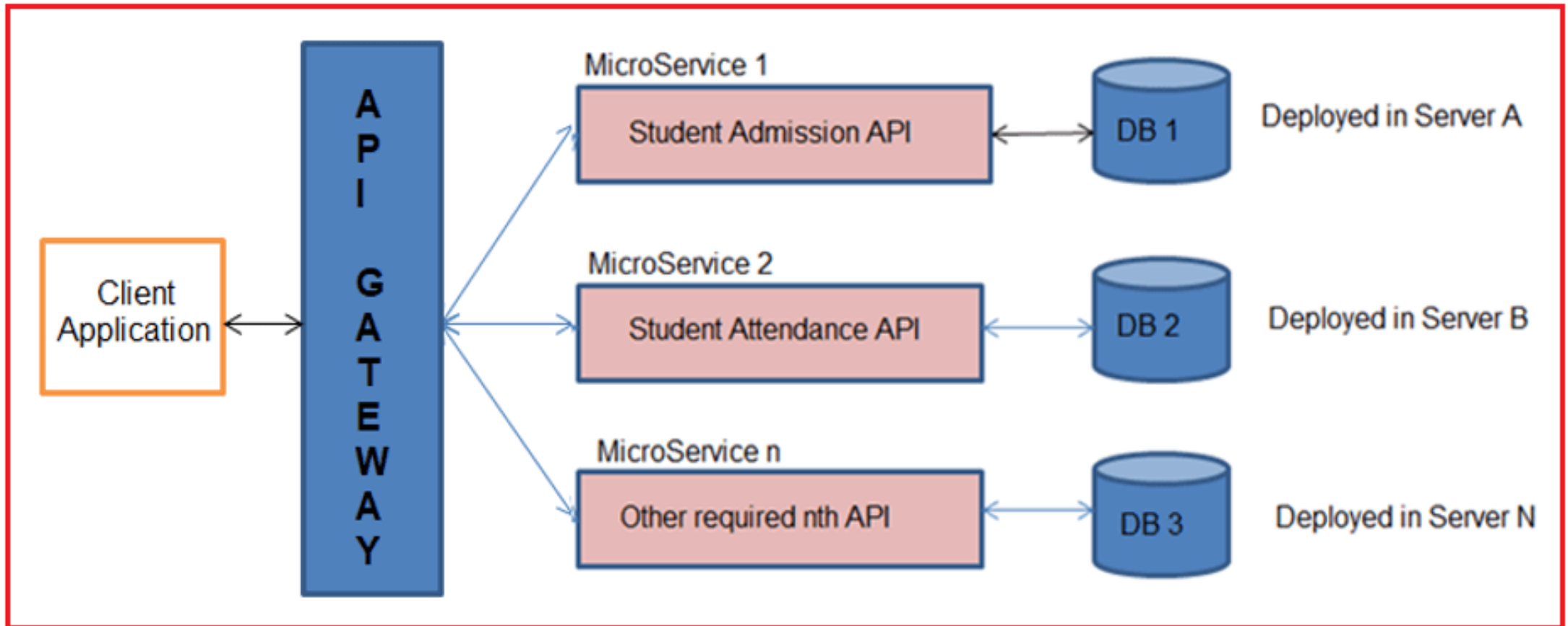
l'architecture monolithique



Architecture de Microservices



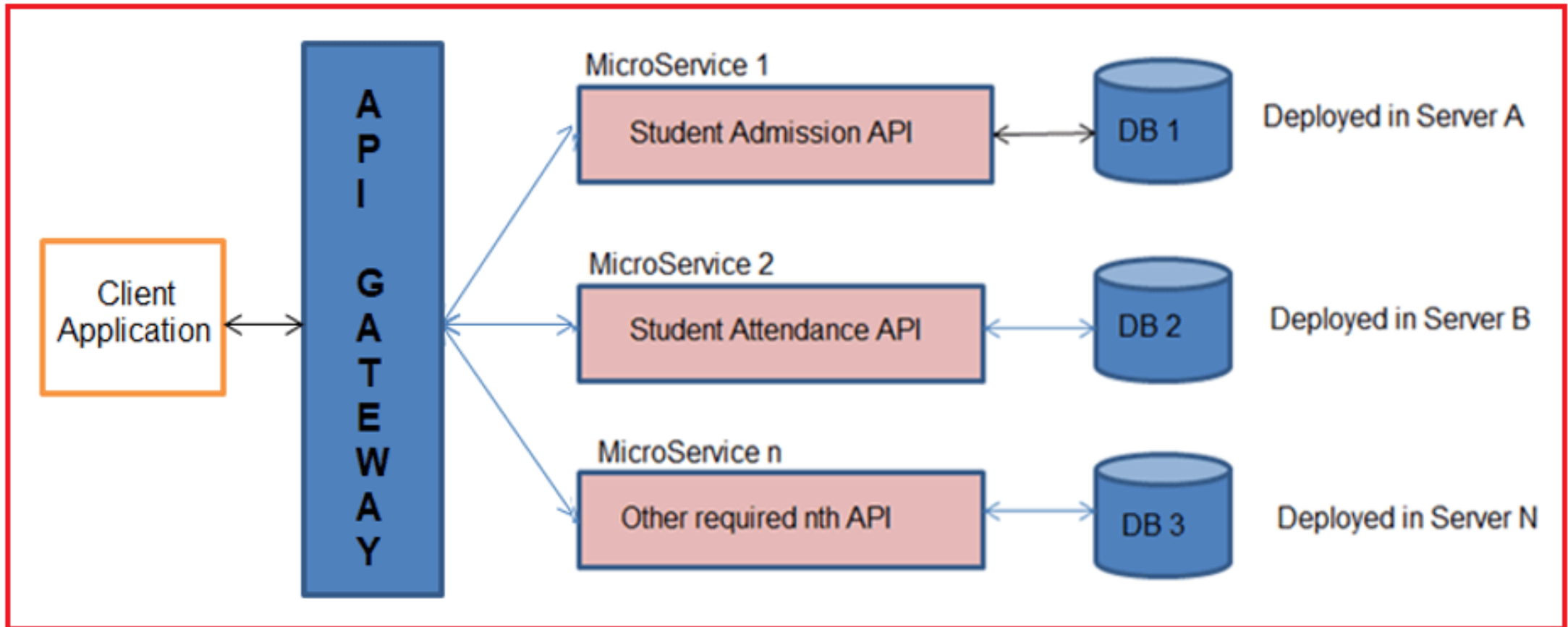
L'API Gateway



L'API Gateway Ocelot

- L'API Gateway n'est rien d'autre qu'une couche middleware qui dirige les appels de requête HTTP entrants des applications clientes vers un microservice spécifique sans exposer directement les détails du microservice au client et renvoyer les réponses générées à partir du microservice respectif.
- Ocelot est une passerelle API Open Source pour la plate-forme .NET/Core qui est officiellement prise en charge par Microsoft.
- Ocelot est également largement utilisé par Microsoft et d'autres géants de la technologie pour la gestion des microservices.
- La dernière version d'ocelot ne prend en charge que les applications .NET Core basées sur la version 3.1 et supérieure.

L'API Gateway



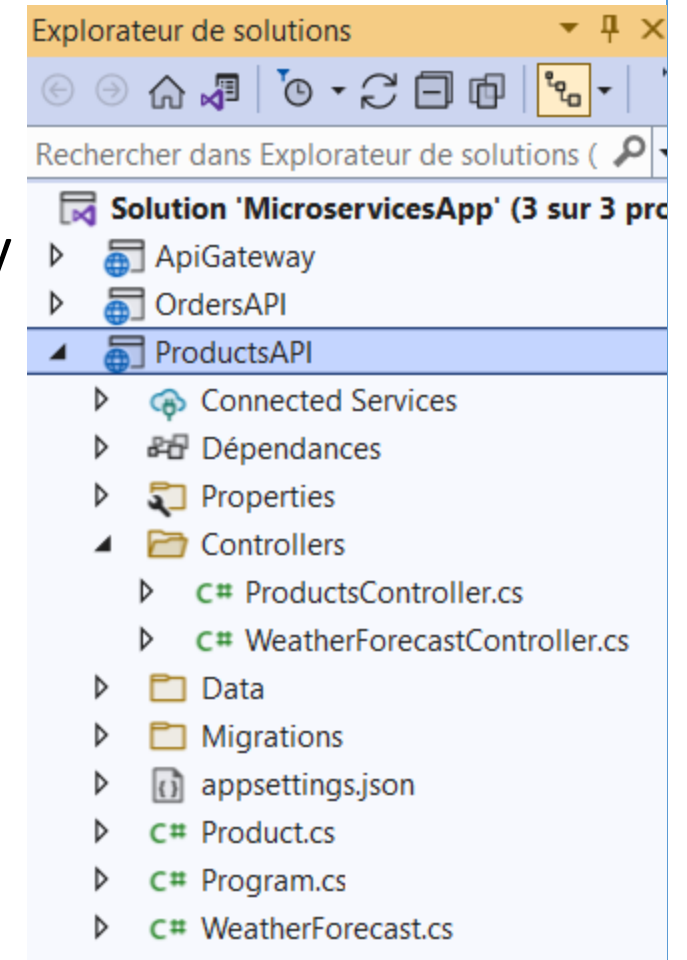
Fonctionnalités de la passerelle API

- **Virtualisation des API** : les passerelles API agissent comme un point d'entrée unique pour tous les microservices configurés, évitent la disponibilité directe des microservices pour les clients et masquent les détails de version des microservices.
- **Sert de couche supplémentaire de microservices de sécurité** : les passerelles API empêchent les attaques malveillantes en fournissant une couche supplémentaire de protection contre les vecteurs d'attaque et les pirates tels que l'injection SQL, les exploits de l'analyseur XML et les attaques par déni de service (DoS), et les soumissions de données de formulaire falsifiées.
- **Diminution de la complexité des microservices** : les techniques d'autorisation telles que JWT et d'autres préoccupations de développement peuvent constituer plus de temps pour le développement de chaque microservice. Une passerelle API peut gérer ces problèmes par elle-même et supprime la charge de développement de votre code API.

Création de Microservices ASP.NET Core

Dans ce qui suit on va créer une solution contenant 3 projets ASP.NET Core WebAPI.

- Le 1^{er} projet est un microservice nommé ProductsAPI
- Le 2^{ème} projet est un microservice nommé OrdersAPI
- Le 3^{ème} projet est le service Gateway nommé ApiGateway



Microservice1 ProductsAPI

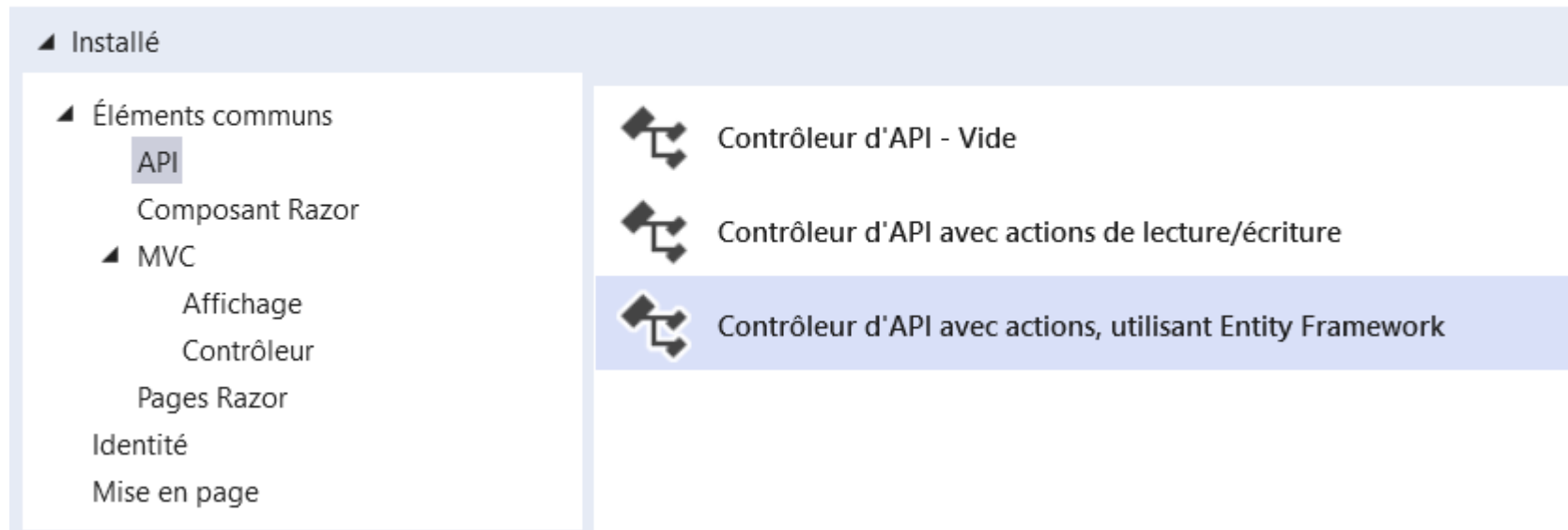
- Créer un projet ASP.NET Core Web API nommé ProductsAPI
- Ajouter la classe Products suivante

```
namespace ProductsAPI
{
    public class Product
    {
        public int ProductId { get; set; }
        public string Name { get; set; } = String.Empty;
        public int StockQte { get; set; }
        public double Price { get; set; }
    }
}
```

Microservice1 ProductsAPI

- Dans le dossier Controllers, Ajouter un nouveau contrôleur API nommé ProductsController:

Ajouter un nouvel élément généré automatiquement



Microservice1 ProductsAPI

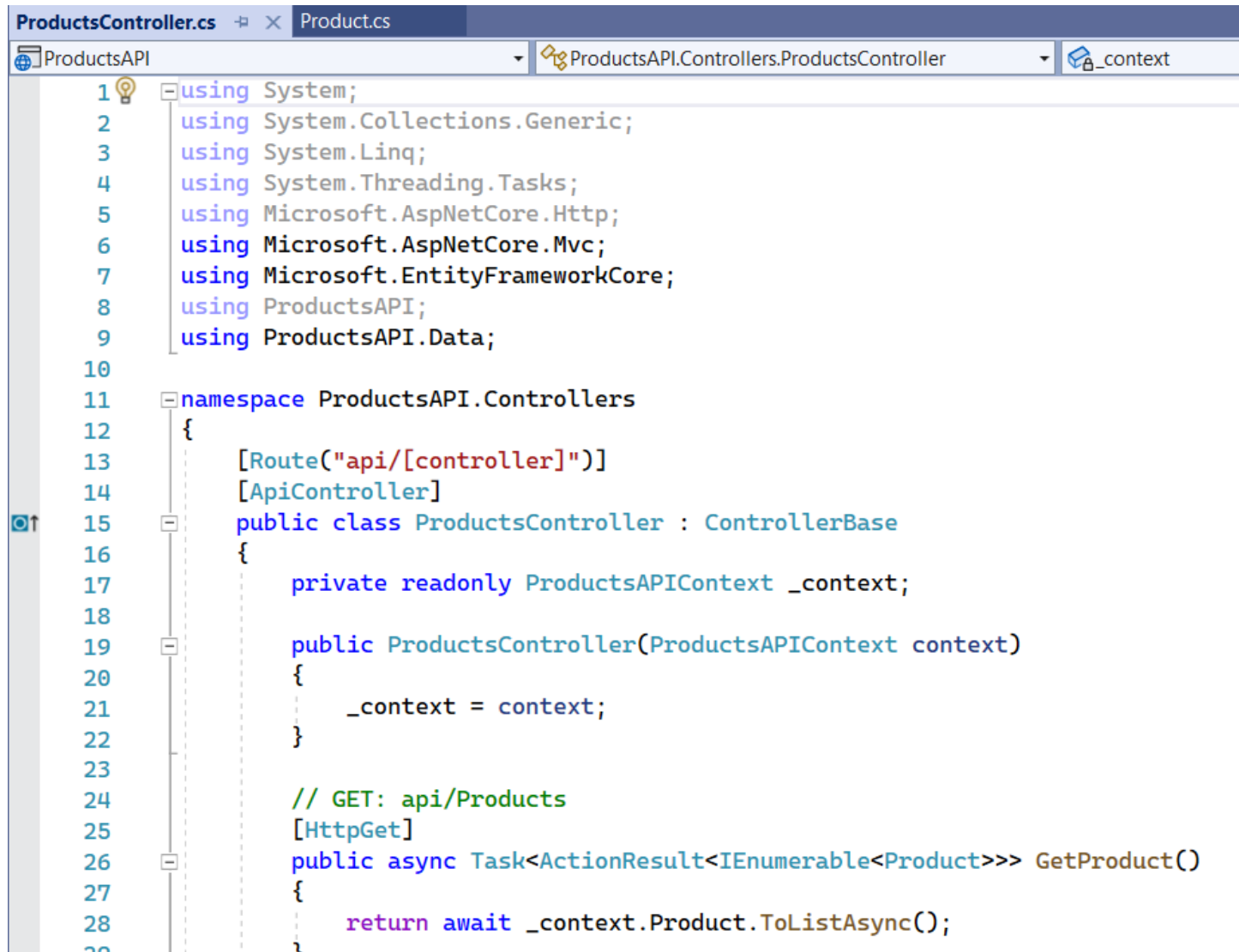
- Choisir Product comme classe modèle et créer une nouvelle classe de contexte.
- Donner le nom du contrôleur puis cliquez sur Ajouter.

×

Ajouter Contrôleur d'API avec actions, utilisant Entity Framew

Classe de modèle	<input type="text" value="Product (ProductsAPI)"/>
Classe de contexte de données	<input type="text" value="ProductsAPIContext (ProductsAPI.Data)"/> <input data-bbox="1847 868 1923 925" type="button" value="+"/>
Nom du contrôleur	<input type="text" value="ProductsController"/>

Microservice1 ProductsAPI

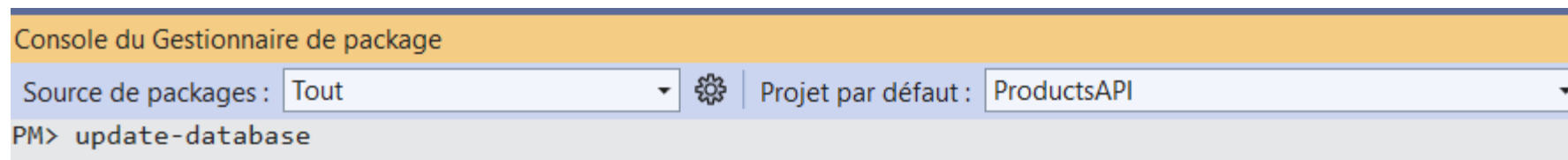
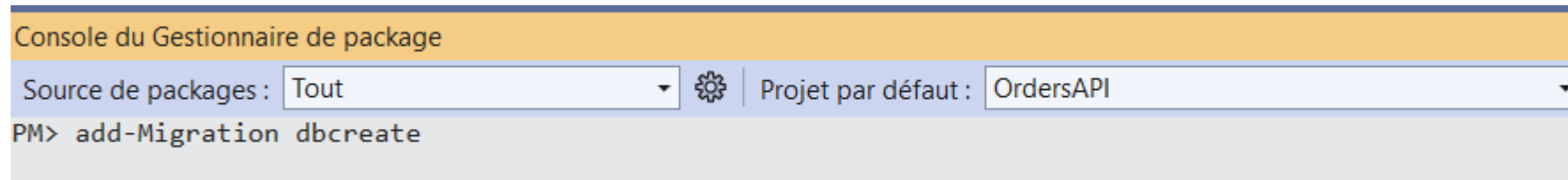


The screenshot shows the Visual Studio IDE with the 'ProductsController.cs' file open. The file is part of the 'ProductsAPI' project, specifically within the 'ProductsAPI.Controllers' namespace. The code defines a 'ProductsController' class that inherits from 'ControllerBase'. It includes several 'using' statements at the top for various .NET and ASP.NET Core namespaces. The class has a private readonly field '_context' of type 'ProductsAPIContext', which is initialized in the constructor. A single action method 'GetProduct()' is shown, which is an asynchronous GET endpoint that returns a list of products from the context.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Threading.Tasks;
5 using Microsoft.AspNetCore.Http;
6 using Microsoft.AspNetCore.Mvc;
7 using Microsoft.EntityFrameworkCore;
8 using ProductsAPI;
9 using ProductsAPI.Data;
10
11 namespace ProductsAPI.Controllers
12 {
13     [Route("api/[controller]")]
14     [ApiController]
15     public class ProductsController : ControllerBase
16     {
17         private readonly ProductsAPIContext _context;
18
19         public ProductsController(ProductsAPIContext context)
20         {
21             _context = context;
22         }
23
24         // GET: api/Products
25         [HttpGet]
26         public async Task<ActionResult<IEnumerable<Product>>> GetProduct()
27         {
28             return await _context.Product.ToListAsync();
29         }
30     }
31 }
```

Microservice1 ProductsAPI

- Avant de tester notre contrôleur API, il faut lancer une migration pour créer la base de données.
- N'oubliez pas de choisir le projet ProductsAPI comme projet par défaut dans la console de gestionnaire de package.



Test du Microservice ProductsAPI

Swagger UI

https://localhost:7066/swagger/index.html

Swagger
Supported by SMARTBEAR

Select a definition: ProductsAPI v1

ProductsAPI 1.0 OAS3

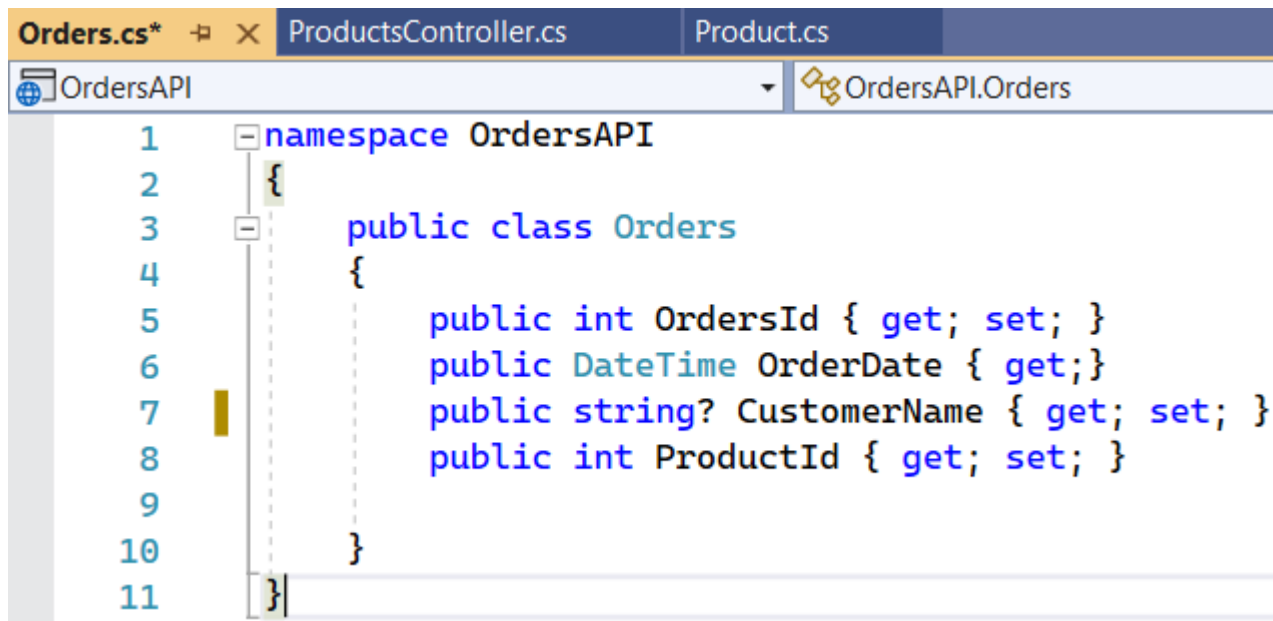
<https://localhost:7066/swagger/v1/swagger.json>

Products

- GET /api/Products
- POST /api/Products
- GET /api/Products/{id}
- PUT /api/Products/{id}

Microservice 2 OrdersAPI

- Créer un projet ASP.NET Core Web API nommé OrdersAPI
- Ajouter la classe Orders suivante

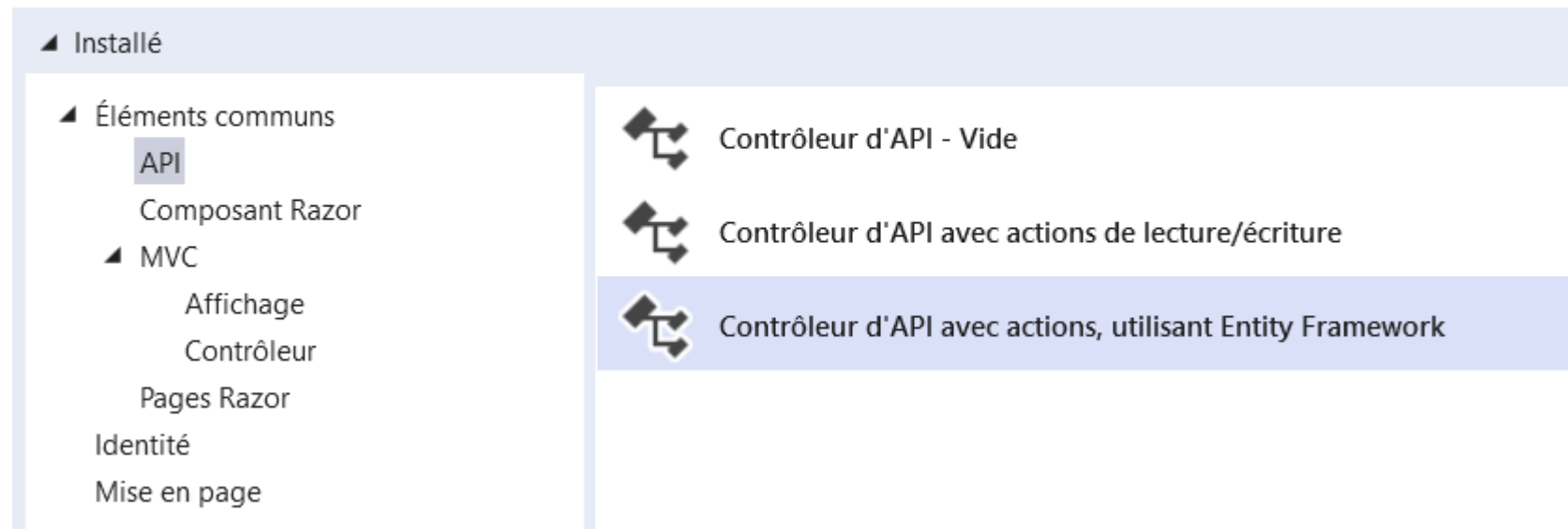


```
1 namespace OrdersAPI
2 {
3     public class Orders
4     {
5         public int OrdersId { get; set; }
6         public DateTime OrderDate { get; }
7         public string? CustomerName { get; set; }
8         public int ProductId { get; set; }
9     }
10 }
11 }
```

Microservice2 OrdersAPI

- Dans le dossier Controllers, Ajouter un nouveau contrôleur API nommé OrdersController:

Ajouter un nouvel élément généré automatiquement



Microservice2 OrdersAPI

- Choisir Orders comme classe modèle et créer une nouvelle classe de contexte.
- Donner le nom du contrôleur puis cliquez sur Ajouter.

×

Ajouter Contrôleur d'API avec actions, utilisant Entity Framew

Classe de modèle

Orders (OrdersAPI) ▾

Classe de contexte de données

OrdersAPIContext (OrdersAPI.Data) ▾

+

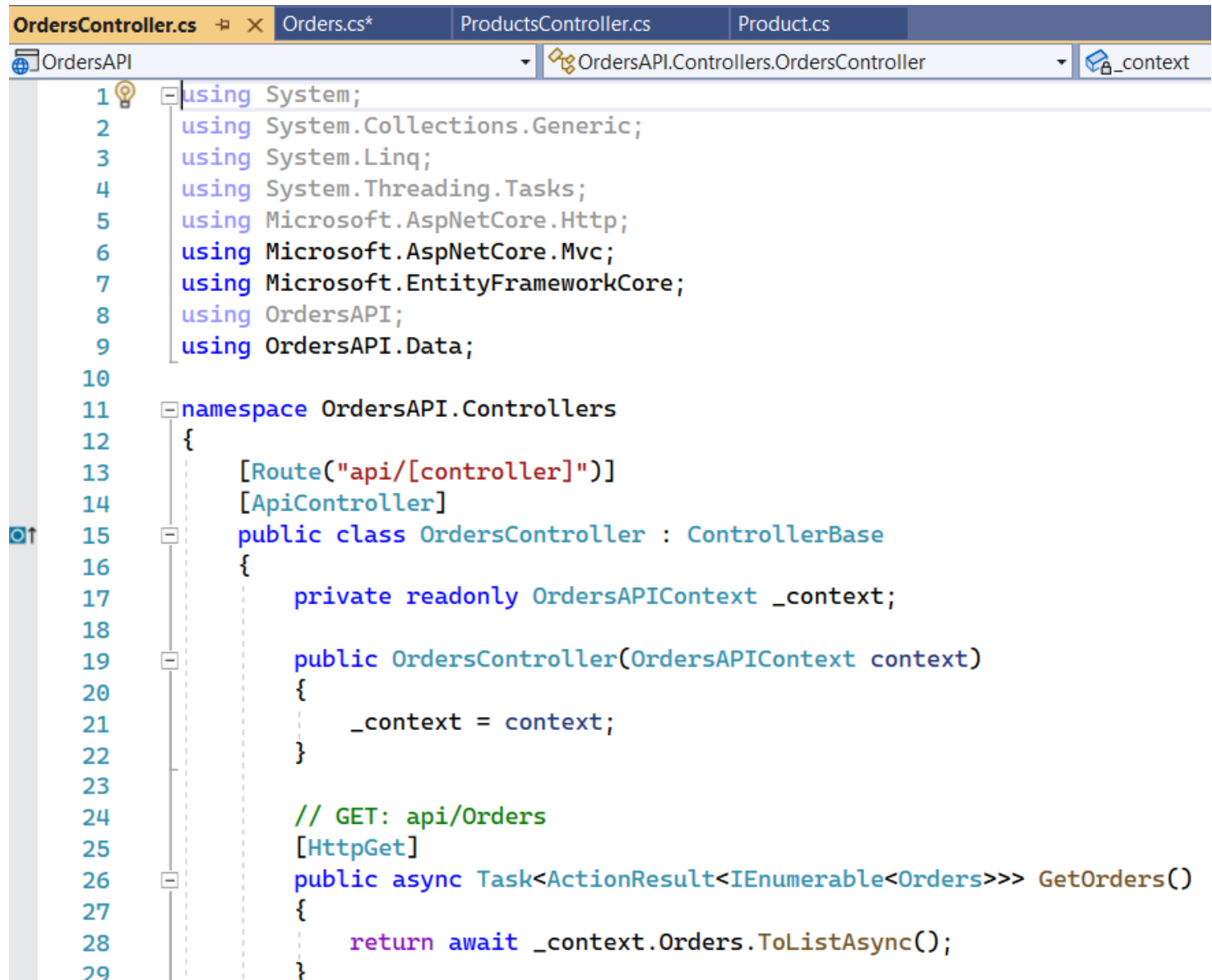
Nom du contrôleur

OrdersController

Ajouter

Annuler

Microservice2 OrdersAPI

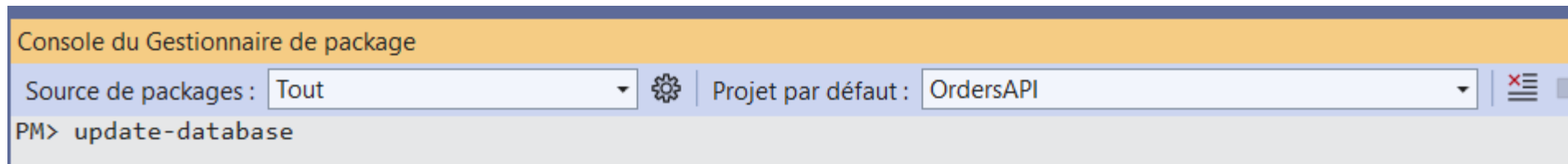
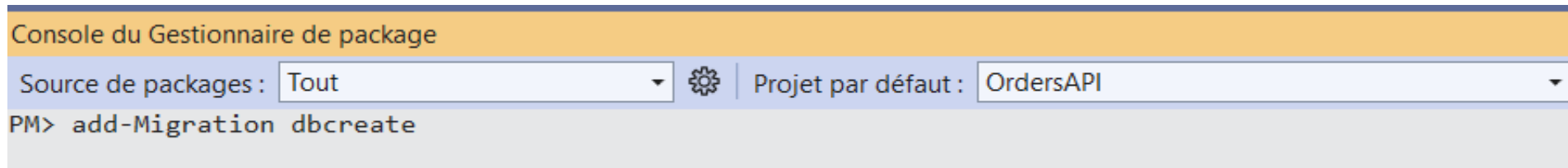


```
OrdersController.cs  Orders.cs*  ProductsController.cs  Product.cs
OrdersAPI  OrdersAPI.Controllers.OrdersController  _context

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using Microsoft.AspNetCore.Http;
6  using Microsoft.AspNetCore.Mvc;
7  using Microsoft.EntityFrameworkCore;
8  using OrdersAPI;
9  using OrdersAPI.Data;
10
11 namespace OrdersAPI.Controllers
12 {
13     [Route("api/[controller]")]
14     [ApiController]
15     public class OrdersController : ControllerBase
16     {
17         private readonly OrdersAPIContext _context;
18
19         public OrdersController(OrdersAPIContext context)
20         {
21             _context = context;
22         }
23
24         // GET: api/Orders
25         [HttpGet]
26         public async Task<ActionResult<IEnumerable<Orders>>> GetOrders()
27         {
28             return await _context.Orders.ToListAsync();
29         }
30     }
31 }
```

Microservice2 OrdersAPI

- Avant de tester notre contrôleur API, il faut lancer une migration pour créer la base de données.
- N'oubliez pas de choisir le projet OrdersAPI comme projet par défaut dans la console de gestionnaire de package.



Test du Microservice OrdersAPI

Swagger UI

https://localhost:7065/swagger/index.html

Swagger
Supported by SMARTBEAR

Select a definition: OrdersAPI v1

OrdersAPI ^{1.0} OAS3

<https://localhost:7065/swagger/v1/swagger.json>

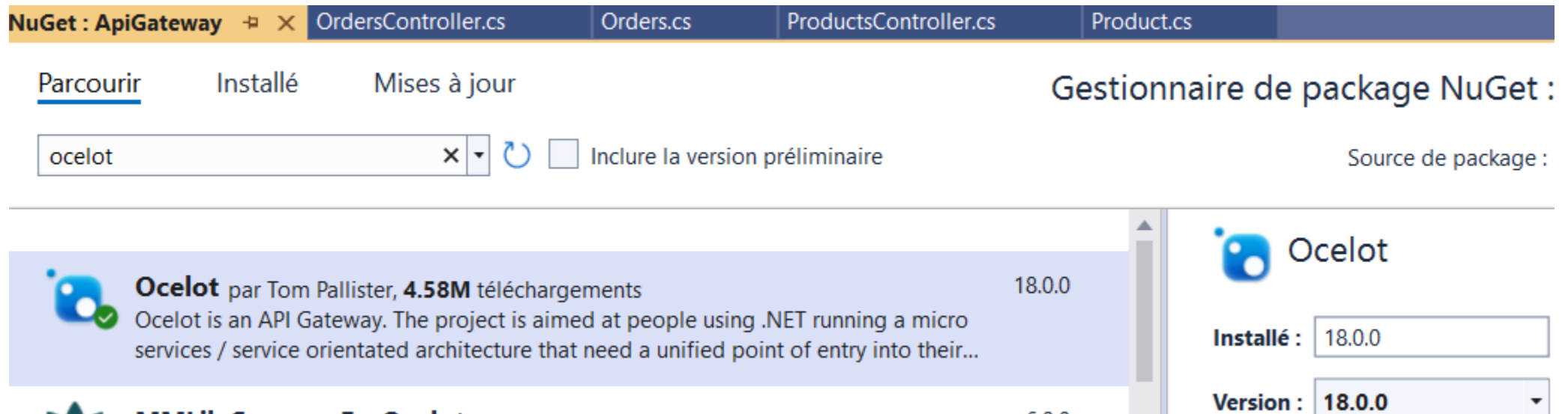
Orders

- GET /api/Orders
- POST /api/Orders
- GET /api/Orders/{id}
- PUT /api/Orders/{id}

Création de l'API Gateway

Dans la même solution:

- Créer un 3^{ème} projet ASP.NET Core web API nommé ApiGateway.
- Dans ce projet installer le gateway Ocelot.
- Dans gestionnaire de package Nuget chercher Ocelot 18.0.0



Le fichier Ocelot.json

- Dans la racine du projet créer un fichier json nommé Ocelot.json
- Dans ce fichier, nous allons mapper nos URL de microservice avec nos URL d'application Gateway Ocelot.


```
1 {
2   "Routes": [
3     {
4       "DownstreamPathTemplate": "/api/Products",
5       "DownstreamScheme": "https",
6       "DownstreamHostAndPorts": [
7         {
8           "Host": "localhost",
9           "Port": 7066
10        }
11      ],
12       "UpstreamPathTemplate": "/apigateway/Products",
13       "UpstreamHttpMethod": [ "GET", "PUT", "POST" ]
14     },
15     {
16       "DownstreamPathTemplate": "/api/Orders",
17       "DownstreamScheme": "https",
18       "DownstreamHostAndPorts": [
19         {
20           "Host": "localhost",
21           "Port": 7065
22        }
23      ],
24       "UpstreamPathTemplate": "/apigateway/Orders",
25       "UpstreamHttpMethod": [ "GET", "PUT", "POST" ]
26     },
27     {
28       "DownstreamPathTemplate": "/api/Products/{id}",
29       "DownstreamScheme": "https",
30       "DownstreamHostAndPorts": [
```

```
31     {
32       "Host": "localhost",
33       "Port": 7066
34     }
35   ],
36   "UpstreamPathTemplate": "/apigateway/products/{id}",
37   "UpstreamHttpMethod": [ "Get", "Post" ]
38 },
39 {
40   "DownstreamPathTemplate": "/api/Orders/{id}",
41   "DownstreamScheme": "https",
42   "DownstreamHostAndPorts": [
43     {
44       "Host": "localhost",
45       "Port": 7065
46     }
47   ],
48   "UpstreamPathTemplate": "/apigateway/Orders/{id}",
49   "UpstreamHttpMethod": [ "Get", "Post" ]
50 },
51 ]
52 }
```

Le fichier Ocelot.json

- La propriété 'Routes' est un type de tableau où nous allons ajouter nos mappages d'URL.
- Le 'DownstreamPathTemplate' est notre point de terminaison de microservice.
- Le 'DownstreamScheme' : le microservice est 'HTTPS' ou 'HTTP'.
- Le 'DownstreamHostAndPorts' : où nous devons définir notre microservice HostName et PortNumber.
- Le 'UpStreamPathTemplate' est une URL ocelot qui est une URL alia pour notre 'DownStreamPathTemlate'.
- Le 'UpstreamHttpMethod' spécifie les méthodes prises en charge.

Intégrer Ocelot Pipeline

- Dans le fichier Program.cs
- ajouter les services et middleware suivants :

```
Program.cs*  Ocelot.json
ApiGateway
1  using Ocelot.DependencyInjection;
2  using Ocelot.Middleware;
3
4  var builder = WebApplication.CreateBuilder(args);
5
6  // Add services to the container.
7
8  builder.Services.AddControllers();
9  // Learn more about configuring Swagger/OpenAPI at
10 builder.Services.AddEndpointsApiExplorer();
11 builder.Services.AddSwaggerGen();
12
13 // services à ajouter
14 builder.Configuration.AddJsonFile("Ocelot.json");
15 builder.Services.AddOcelot();
16
17 var app = builder.Build();
18
19 // Configure the HTTP request pipeline.
20 if (app.Environment.IsDevelopment())
21 {
22     app.UseSwagger();
23     app.UseSwaggerUI();
24 }
25
26 app.UseHttpsRedirection();
27 // middleware à ajouter
28 app.UseOcelot().Wait();
29
30 app.UseAuthorization();
31
32 app.MapControllers();
```

Test des Microservices via l'API Gateway

GET https://localhost:7162/ ●

GET https://localhost:7162/ ●

PUT https://localhost:7162/ ●

PUT https://localhost:7162/ ●

+

...

No Environment

▼

https://localhost:7162/apigateway/Products

Save ▼

No Environment

GET ▼

https://localhost:7162/apigateway/Products

Send ▼

Params

Authorization

Headers (7)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
	Key	Value	Description		

Body

Cookies

Headers (4)

Test Results

Status: 200 OK

Time: 2.81 s

Size: 258 B

Save Response ▼

Pretty

Raw

Preview

Visualize

JSON ▼

≡

```
1  [
2    {
3      "productId": 1,
4      "name": "prod1",
5      "stockQte": 50,
6      "price": 1000
7    },
8    {
9      "productId": 2,
10     "name": "prod2",
11     "stockQte": 40,
12     "price": 1520
13   }
```

Test des Microservices via l'API Gateway

GET https://localhost:7162/

GET https://localhost:7162/

PUT https://localhost:7162/

PUT https://localhost:7162/

+

...

No Environment

▼

https://localhost:7162/apigateway/Orders

Save

▼

GET

▼

https://localhost:7162/apigateway/Orders

Send

▼

Params

Authorization

Headers (7)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
	Key	Value	Description		

Body

Cookies

Headers (4)

Test Results

Status: 200 OK

Time: 1617 ms

Size: 303 B

Save Response

▼

Pretty

Raw

Preview

Visualize

JSON

▼

```
1  [
2    {
3      "ordersId": 1,
4      "orderDate": "0001-01-01T00:00:00",
5      "customerName": 5,
6      "productId": 1
7    },
8    {
9      "ordersId": 2,
10     "orderDate": "0001-01-01T00:00:00",
11     "customerName": 10,
12     "productId": 1
13   }
```

Test des Microservices via l'API Gateway

GET https://localhost:7162/ ●

GET https://localhost:7162/ ●

PUT https://localhost:7162/ ●

PUT https://localhost:7162/ ●

+

...

No Environment ▼

https://localhost:7162/apigateway/Products/1

Save ▼

GET ▼

https://localhost:7162/apigateway/Products/1

Send ▼

Params

Authorization

Headers (7)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
	Key	Value	Description		

Body

Cookies

Headers (4)

Test Results

Status: 200 OK

Time: 303 ms

Size: 197 B

Save Response ▼

Pretty

Raw

Preview

Visualize

JSON ▼

1

{

2

"productId": 1,

3

"name": "prod1",

4

"stockQte": 50,

5

"price": 1000

6

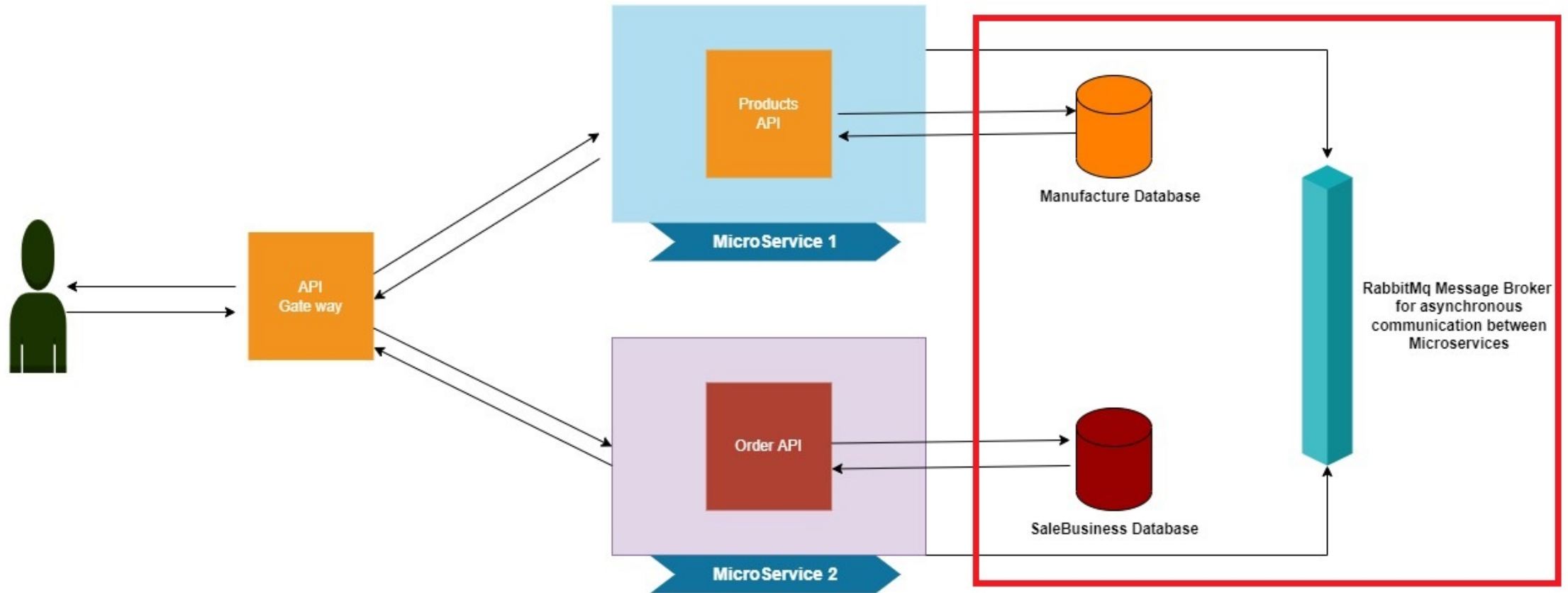
}

Communication de données asynchrones entre microservices à l'aide de RabbitMQ Message Broker avec MassTransit

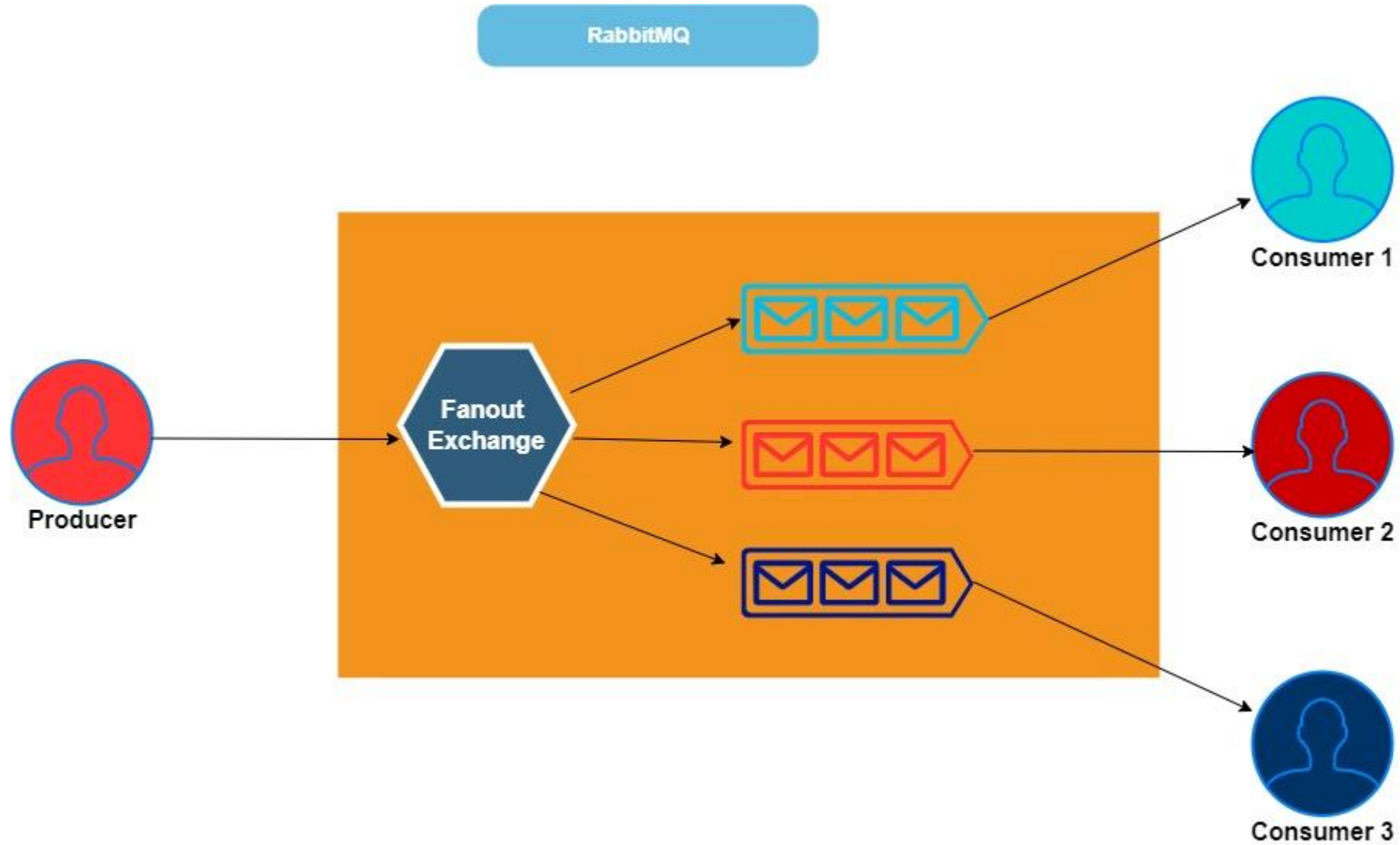
Rabbit MQ

- RabbitMQ est un diffuseur de messages qui permet de partager des données de manière asynchrone entre les émetteurs et les récepteurs.
- On constate bien que nous n'avons pas d'informations 'Produit' dans le microservices 'Orders'. Nous devons donc trouver un moyen d'enregistrer les informations 'Produit' requises dans le microservice Orders à partir du microservice Products.
- Après l'ajout d'un nouveau produit dans ProductsAPI, on va enregistrer les informations nécessaire dans une table produit du microservice OrdersAPI, cette deuxième table contiendra uniquement les information requises par OrdersAPI.
- Avec Rabbit MQ on pourra assurer une communication entre les deux microservices.

Rabbit MQ



Rabbit MQ



MassTransit

- MassTransit est un logiciel open source gratuit qui peut envelopper les «Message Brokers » tels que « RabbitMQ », « Azure Service Bus », « SQS », « ActiveMQ Service Bus », etc
- MassTransit fournit un moyen simple de configurer les messages dans nos applications .NET.

Installer RabbitMQ

- Installez Docker Desktop sur votre ordinateur local. (<https://www.docker.com/products/docker-desktop/>)
- Vérifier que docker fonctionne correctement.
- Télécharger une image docker de RabbitMQ en suivant les étapes suivantes:

```
Administrateur : Invite de commandes
Microsoft Windows [version 10.0.19044.2130]
(c) Microsoft Corporation. Tous droits réservés.

C:\Windows\system32>cd/

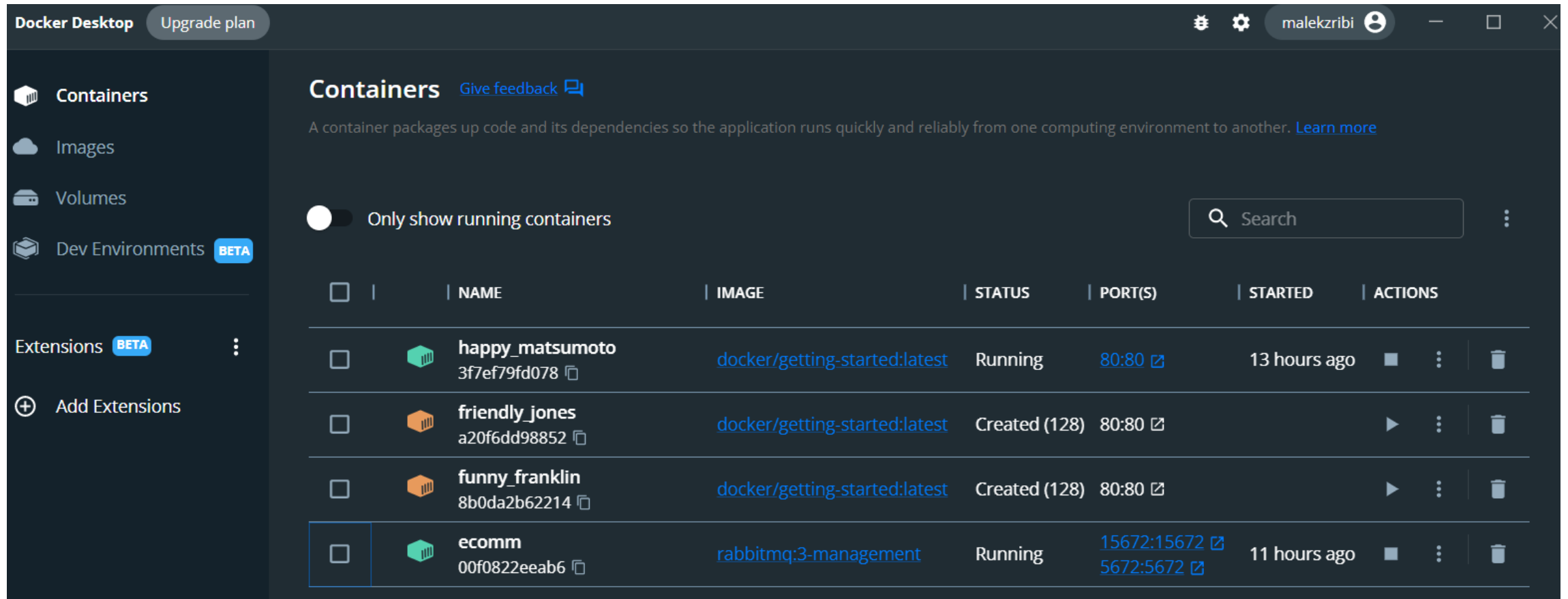
C:\>docker run -d --hostname my-rabbit --name ecomm -p 15672:15672 -p 5672:5672 rabbitmq:3-management
Unable to find image 'rabbitmq:3-management' locally
3-management: Pulling from library/rabbitmq
eaead16dc43b: Pull complete
d5e775568c00: Pull complete
9300f4c930ff: Pull complete
c408cd7bb376: Pull complete
9e00bcf620de: Pull complete
5029678183b6: Pull complete
8a316bbf4e78: Pull complete
17bf36dd12be: Pull complete
a433895f52d5: Pull complete
5253681bc2c3: Pull complete
Digest: sha256:a13c9c763900dad14e62a3137933ec70a8d7cb7c4971f6ad93e4a471c433921
Status: Downloaded newer image for rabbitmq:3-management
00f0822eeab6d544888787c7361c991776dfe394c6e87e59276226d3c714deb7

C:\>
```

Changer ce 1^{er}
numéro de port vers
4001

Installer RabbitMQ

- Vérifier que l'image apparaît et en cours d'exécution dans docker desktop

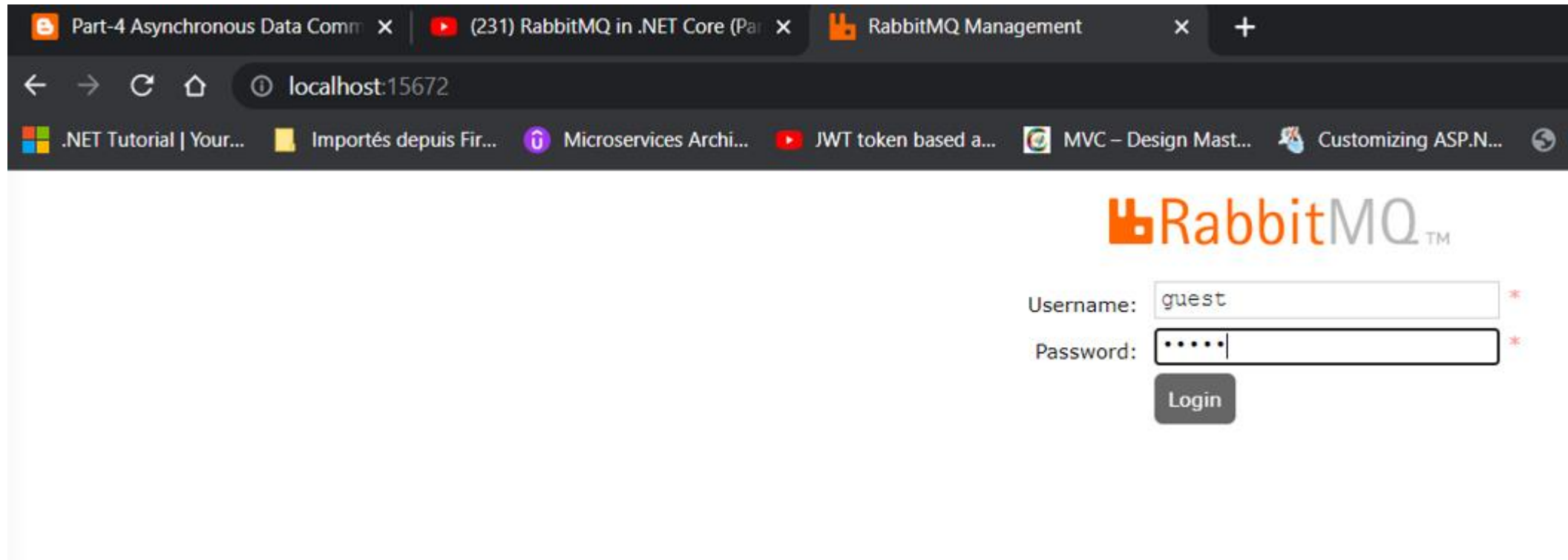


The screenshot shows the Docker Desktop application window. On the left sidebar, the 'Containers' tab is selected. The main area displays a list of containers. A toggle switch for 'Only show running containers' is turned on. A search bar is present in the top right of the container list. The container list has columns for NAME, IMAGE, STATUS, PORT(S), STARTED, and ACTIONS. The 'ecommm' container is highlighted with a blue border and is in a 'Running' state.

	NAME	IMAGE	STATUS	PORT(S)	STARTED	ACTIONS
<input type="checkbox"/>	happy_matsumoto 3f7ef79fd078	docker/getting-started:latest	Running	80:80	13 hours ago	
<input type="checkbox"/>	friendly_jones a20f6dd98852	docker/getting-started:latest	Created (128)	80:80		
<input type="checkbox"/>	funny_franklin 8b0da2b62214	docker/getting-started:latest	Created (128)	80:80		
<input checked="" type="checkbox"/>	ecommm 00f0822eeab6	rabbitmq:3-management	Running	15672:15672 5672:5672	11 hours ago	

Installer RabbitMQ

- Lancer le navigateur et taper l'URL suivante pour afficher la page d'administration de rabbit.
- Pour se connecter username: guest et password: guest



Installer RabbitMQ

The screenshot shows the RabbitMQ Management interface in a web browser. The browser's address bar displays `localhost:15672/#/queues`. The page header includes the RabbitMQ logo, version information (RabbitMQ 3.11.2, Erlang 25.1.2), and a refresh status (Refreshed 2022-10-26 23:06:49). The navigation menu at the top has tabs for Overview, Connections, Channels, Exchanges, Queues (which is active), and Admin. On the right side of the header, there are controls for the virtual host (set to 'All') and the user (set to 'guest' with a 'Log out' button). The main content area is titled 'Queues' and shows a dropdown for 'All queues (0)'. Below this, a pagination section indicates 'Page 1 of 0' and 'Filter:'. A message states '... no queues ...'. At the bottom of the main area, there is a link to 'Add a new queue'. The footer contains a list of links: HTTP API, Server Docs, Tutorials, Community Support, Community Slack, Commercial Support, Plugins, GitHub, and Changelog.

Part-4 Asynchronous Data Comm x | (231) RabbitMQ in .NET Core (Par x | RabbitMQ Management x +

localhost:15672/#/queues

Refreshed 2022-10-26 23:06:49 Refresh every 5 seconds

Virtual host All

Cluster rabbit@my-rabbit

User guest Log out

Queues

▼ All queues (0)

Pagination

Page 1 of 0 - Filter: ☐ Regex ?

Displaying 0 item , page size up to: 100

... no queues ...

► Add a new queue

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

RabbitMQ

- Dans notre solution Créer une bibliothèque de classes nommé Shared contenant les classes DTO pour gérer les messages RabbitMQ.
- Dans notre solution on a deux microservices l'un agira en tant qu'éditeur et l'autre agira en tant que consommateur. Ainsi, le Type du message sera un modèle, une classe qui est créée dans la Bibliothèque de classes et qui est consommée par nos deux applications Microservices.
- Ajoutez la référence de la bibliothèque de classes créée dans les deux microservices

```
namespace Shared.Models
{
    public class ProductCreated
    {
        public int Id { get; set; }
        public string Name { get; set; } =String.Empty;
    }
}
```


Installation des packages NuGet

- Pour configurer RabbitMQ et MassTransit, nous devons installer les packages NuGet suivants dans les deux projets microservices ProductsAPI et OrdersAPI:
 - 1/ MassTransit
 - 2/ MassTransit.AspNetCore
 - 3/ MassTransit.RabbitMQ

Code côté Microservice Emetteur

- Dans le Microservice ProductsAPI qui est l'application émetteur configurez les services 'MassTransit' et 'RabbitMQ' dans le fichier Program.cs
- (Ligne : 1) Inscrit le service « MassTransit ».
- (Ligne : 2) Le service 'RabbitMQ' est configuré à l'intérieur du service 'MassTransit'.
- (Ligne : 3) Définition de notre hôte 'RabbitMQ'. Ici, le port '4001' est mon port personnalisé exposé depuis le conteneur Docker.
- (Ligne : 4 et 5) Le nom d'utilisateur et le mot de passe par défaut pour 'RabbitMQ' sont 'guest'.

```
builder.Services.AddMassTransit(options => {  
    options.UsingRabbitMq((context, cfg) => {  
        cfg.Host(new Uri("rabbitmq://localhost:4001"), h => {  
            h.Username("guest");  
            h.Password("guest");  
        });  
    });  
});
```

Code côté Microservice Emetteur

- À l'intérieur du constructeur de 'ProductController' injectons le service 'MassTransit.IPublishEndpoint'

```
[Route("api/[controller]")]
[ApiController]
public class ProductsController : ControllerBase
{
    private readonly ProductsAPIContext _context;
    private readonly IPublishEndpoint _publishEndpoint;

    public ProductsController(ProductsAPIContext context, IPublishEndpoint publishEndpoint)
    {
        _context = context;
        _publishEndpoint = publishEndpoint;
    }
}
```

- Maintenant, implémentons la logique de publication dans la méthode d'action "PostProduct" de ProductController.

Code côté Microservice Emetteur

```
[HttpPost]
public async Task<ActionResult<Product>> PostProduct(Product product)
{
    _context.Product.Add(product);
    await _context.SaveChangesAsync();
    await _publishEndpoint.Publish<ProductCreated>(new ProductCreated
    {
        Id = product.ProductId,
        Name = product.Name
    });

    return CreatedAtAction("GetProduct", new { id = product.ProductId }, product);
}
```

- La méthode 'IPublishEndpoint.Publish<T>()' envoie le message dans le 'Fanout Exchange' de RabbitMQ.
- Le type du message est 'ProductCreated'.

Migration et génération de table

- Ajout de la classe Product et génération de la table Products dans OrdersAPI Microservice.

```
1 namespace OrdersAPI
2 {
3     public class Product
4     {
5         public int Id { get; set; }
6         public string Name { get; set; }
7     }
8 }
9
```

- Ajouter l'objet DbSet à la classe de contexte du projet.

```
public class OrdersAPIContext : DbContext
{
    public OrdersAPIContext (DbContextOptions<OrdersAPIContext> options)
        : base(options)
    {
    }

    public DbSet<OrdersAPI.Orders> Orders { get; set; } = default!;
    public DbSet<Product> Products { get; set; }
}
```

Migration et génération de table

- Lancer une Migration pour ajouter la table products à la base de données.

```
Console du Gestionnaire de package
Source de packages : Tout [v] [g] | Projet par défaut : OrdersAPI [v] [x] [≡]
Tapez 'get-help NuGet' pour afficher toutes les commandes NuGet disponibles.
PM> Add-migration addproduct
```

```
PM> update-database
Multiple startup projects set.
Using project 'OrdersAPI' as the startup project.
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Infrastructure[10403]
```

Code côté Microservice récepteur

- OrdersAPI est le Microservice récepteur de message.
- En utilisant le service 'MassTransit.IConsumer<T>', nous allons créer un consommateur de file d'attente RabbitMQ.
- Dans OrdersAPI créer un dossier nommé 'Consumer' puis ajouter une classe nommée 'ProductCreatedConsumer.cs'.

Code côté Microservice récepteur

```
using MassTransit;
using OrdersAPI.Data;
using Shared.Models;

namespace OrdersAPI.Consumer
{
    public class ProductCreatedConsumer : IConsumer<ProductCreated>
    {
        private readonly OrdersAPIContext _OrdersAPIContext;
        public ProductCreatedConsumer(OrdersAPIContext ordersAPIContext)
        {
            _OrdersAPIContext = ordersAPIContext;
        }
        public async Task Consume(ConsumeContext<ProductCreated> context)
        {
            var newProduct = new Product
            {
                //Id = context.Message.Id,
                Name = context.Message.Name
            };
            _OrdersAPIContext.Add(newProduct);
            await _OrdersAPIContext.SaveChangesAsync();
        }
    }
}
```


Code côté Microservice récepteur

- Pour faire de l'entité 'ProductCreatedConsumer' un consommateur de file d'attente RabbitMQ, elle doit hériter de 'MassTransit.IConsumer'.
- la méthode asynchrone 'Consume' implémentée est exécutée à chaque nouveau message reçu par la file d'attente.
- À l'intérieur de cette méthode on met le code pour stocker dans la base les données du message reçu.
- Ensuite dans 'Program.cs', enregistrons les services 'MassTransit' et 'RabbitMQ'.

Code côté Microservice récepteur

```
// Add services to the container.
```

```
builder.Services.AddControllers();
```

```
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
```

```
builder.Services.AddEndpointsApiExplorer();
```

```
builder.Services.AddSwaggerGen();
```

```
builder.Services.AddMassTransit(x => {  
    x.AddConsumer<ProductCreatedConsumer>();  
    x.UsingRabbitMq((context, cfg) => {  
        cfg.Host(new Uri("rabbitmq://localhost:4001"), h => {  
            h.Username("guest");  
            h.Password("guest");  
        });  
        cfg.ReceiveEndpoint("event-listener", e => {  
            e.ConfigureConsumer<ProductCreatedConsumer>(context);  
        });  
    });  
});
```

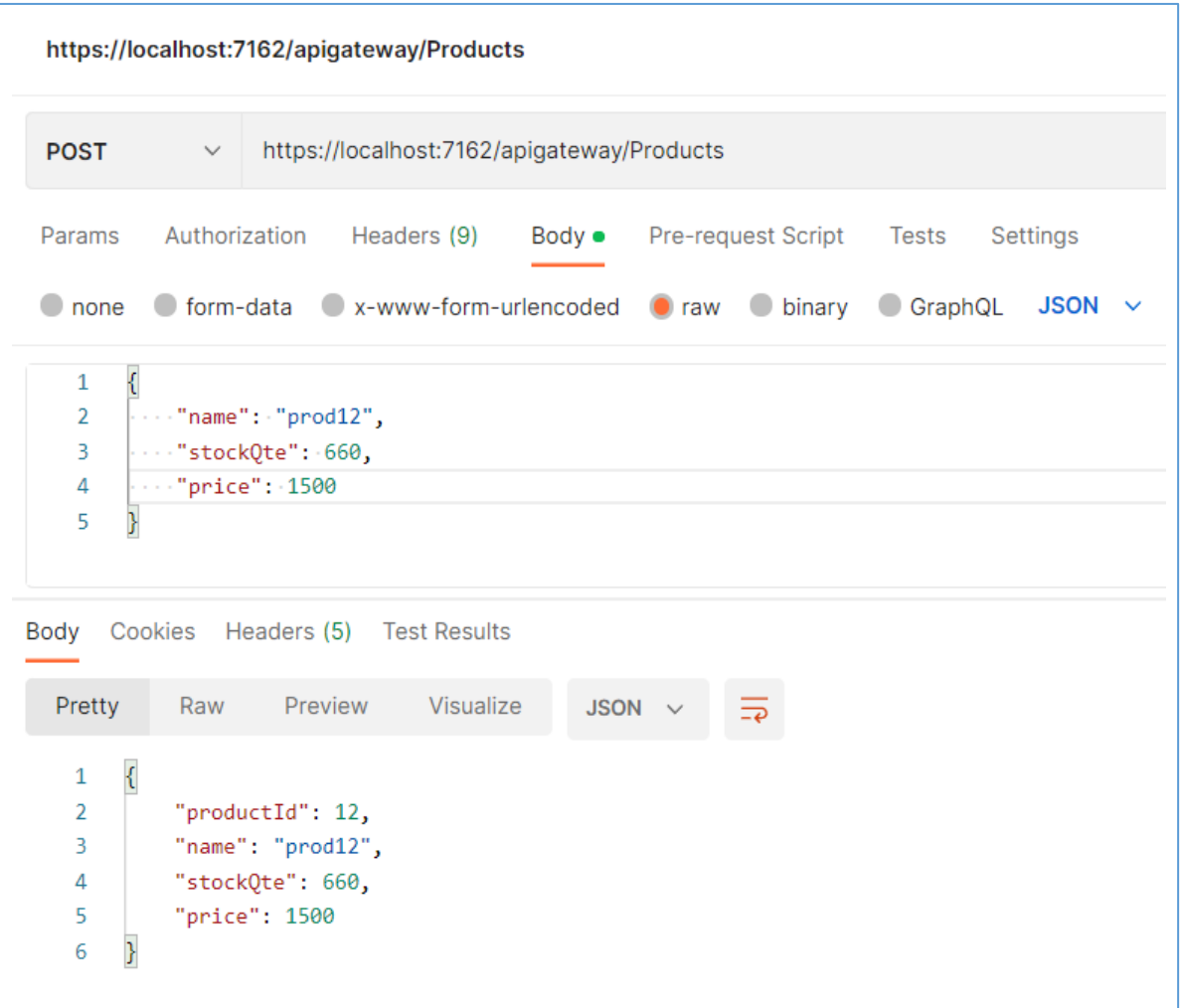
```
var app = builder.Build();
```

Code côté Microservice récepteur

- Dans ce code on définit notre nom de canal ou de file d'attente comme 'event-listener'.
- L'échange Fanout de 'RabbitMQ' pousse les messages vers toutes les files d'attente.
- Aussi on enregistre ici notre entité de canal qui est 'ProductCreatedConsumer' qui écoute chaque nouveau message de la file d'attente RabbitMQ.

Test de la communication entre les Microservices

- Lancer l'exécution des Microservices.
- Lancer docker desktop et vérifier que l'image docker de RabbitMQ est bien en cours d'exécution.
- Lancer postman et envoyer une requête Post pour ajouter un nouveau Prduit dans le microservice ProductAPI.
- Le produit est enregistré aussi dans la table product du Microservice OrdersAPI.
- Ces 2 tables sont synchronisées grâce à la communication asynchrone de RabbitMQ entre les microservices.



Test de la communication entre les Microservices

The screenshot displays two SQL Server Enterprise Manager windows. The background window shows the 'dbo.Products' table with two columns: 'Id' and 'Name'. It contains one data row with 'Id' 1 and 'Name' 'prod12', and a row with NULL values. The foreground window shows the 'dbo.Product' table with four columns: 'ProductId', 'Name', 'StockQte', and 'Price'. It contains 12 data rows, with the 12th row (ProductId 12, Name 'prod12', StockQte 660, Price 1500) highlighted in blue. Both windows have a toolbar with icons for refreshing, filtering, and zooming, and a 'Nombre maximal de lignes' (Maximum number of lines) dropdown set to 1000.

Id	Name
1	prod12
NULL	NULL

ProductId	Name	StockQte	Price
1	prod1	50	1000
2	prod2	40	1520
3	prodmm	30	1600
4	prod12	30	500
5	prodmm	30	1600
6	prodm	30	1600
7	dghdgd	10	110
8	prodm	30	1600
9	prod9	40	16500
10	prod10	40	16500
11	prod11	40	16500
12	prod12	660	1500
NULL	NULL	NULL	NULL