GUIDE DE TRAVAUX PRATIQUES

Topologies de Middleware

d'une Entreprise Engagée

L'importance des Middlewares (API Gateway) dans les Architectures Microservices

Démonstration pratique et mise en situation

Préparé et simulé par :

Mouad BENCAID

Année universitaire : 2025/2026

Table des matières

Table des figures

Introduction

Mes collègues ont déjà traité la partie théorique en expliquant ce que sont les middlewares et leurs différents types. Cette partie va être **assez pratique**.

Je vous ai déjà envoyé ce guide en format PDF. Il contient toutes les étapes nécessaires pour pouvoir suivre avec moi pendant cette démonstration pratique.

Objectif

★ Objectif Principal

L'objectif de cette activité pratique est de vous montrer **l'importance des middlewares** dans un système distribué ou une application microservice. On va se concentrer particulièrement sur le cas de l'**API Gateway**.

À travers cette démonstration, vous allez :

✓ Comprendre les limitations d'une architecture microservices sans API Gateway

✓ Découvrir les avantages apportés par l'introduction d'un API Gateway

✓ Analyser l'impact sur la scalabilité, la résilience et la maintenabilité

✓ Observer concrètement les différences entre les deux approches

Méthodologie

Pendant cette activité, nous allons suivre la stratégie suivante :

1. **Présentation du cas d'étude**
   Je vais présenter un cas d'étude réel afin de vous situer dans le contexte d'une entreprise qui a migré d'une architecture monolithique vers une architecture microservices.

2. **Architecture SANS API Gateway**
   Nous allons déployer localement une application microservice **sans middleware API Gateway**. Nous l'analyserons ensemble et essaierons de comprendre ses limitations principales :
   - Gestion des adresses des services
   - Load balancing manuel
   - Services éphémères
   - Cross-cutting concerns non centralisés

3. **Architecture AVEC API Gateway**
   Ensuite, nous allons déployer localement une deuxième application microservice **avec middleware API Gateway**. Nous examinerons ses avantages et comment elle résout les problèmes identifiés précédemment.

4. **Analyse comparative**
   Enfin, nous comparerons les deux approches pour comprendre concrètement l'impact de l'API Gateway sur l'architecture globale.

! Prérequis

Avant de commencer cette démonstration, assurez-vous d'avoir suivi le guide des prérequis qui vous a été envoyé au préalable. Ce guide contient toutes les instructions nécessaires pour la préparation de votre environnement. Le guide actuel est uniquement destiné à suivre la simulation en temps réel.

Vérifiez que vous avez effectué les étapes suivantes :

- ✓ Installation de Docker et Docker Desktop
- ✓ Clonage des deux repositories (architecture sans Gateway et architecture avec Gateway)
- ✓ Construction des images Docker nécessaires pour les services de chaque architecture

1 Contexte : Cas d'Étude E-commerce

1.1 Situation Initiale : Application Monolithique

Notre cas d'étude concerne une entreprise d'e-commerce qui possédait initialement une **application monolithique**.

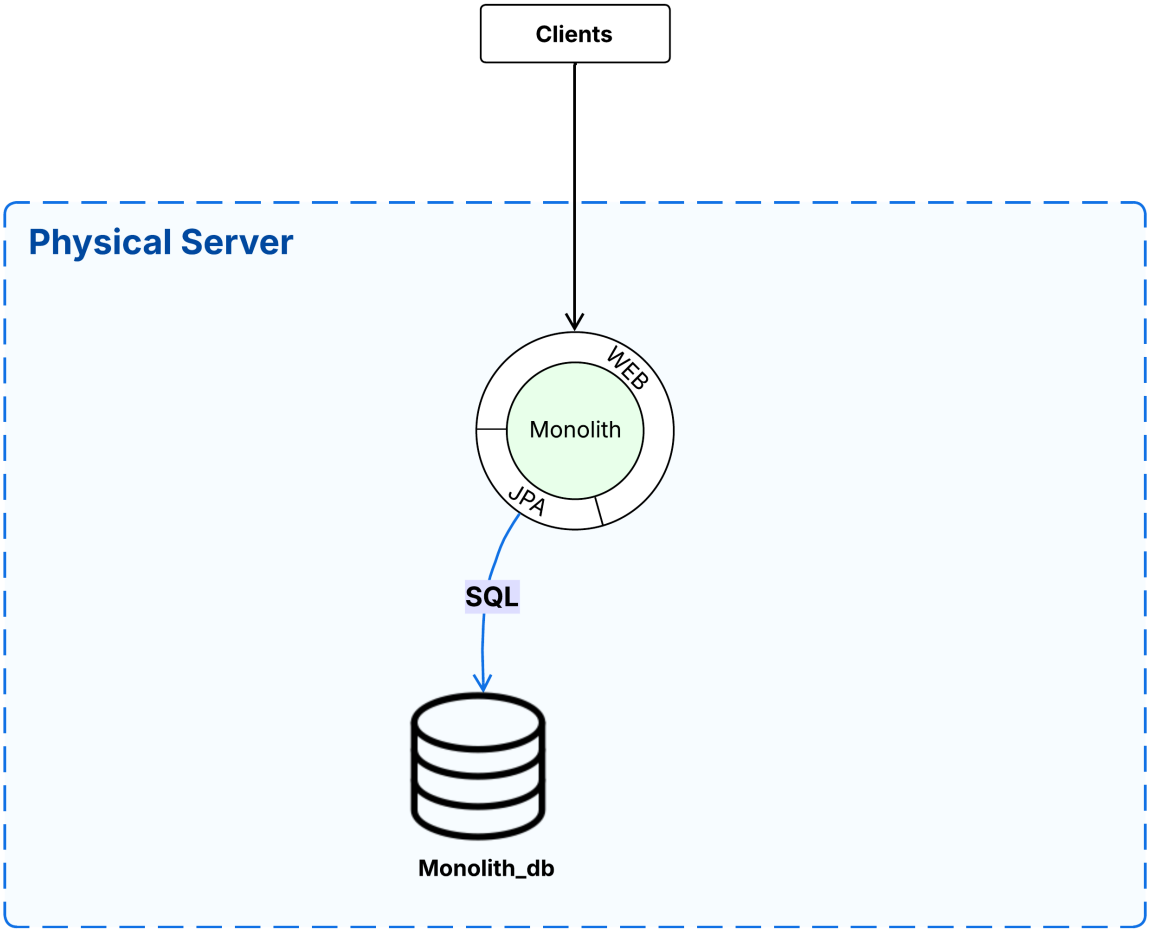

FIGURE 1 – Architecture Monolithique

Au fil du temps, l'entreprise a commencé à rencontrer des difficultés croissantes, notamment avec une scalabilité limitée et une faible tolérance aux pannes.

Limitations de l'architecture monolithique

- L'application était modulaire mais **déployée** dans un seul bloc.
- Les différents modules de l'application ne **pouvaient** pas être scalés **indépendamment** (Couplage Fort).
- Le recours à la scalabilité verticale uniquement (ajout de ressources au serveur).
- Si un module tombe en panne, **toute** l'application **est affectée**.
- Toute modification nécessite le redéploiement complet.

1.2 Décision : Migration vers Microservices

L'entreprise a décidé de faire une migration vers une **architecture distribuée** afin de mieux gérer la charge sur leur infrastructure en tirant profit des avantages de ce type d'architecture comme la **scalabilité horizontale** et **la Tolérance aux pannes**.

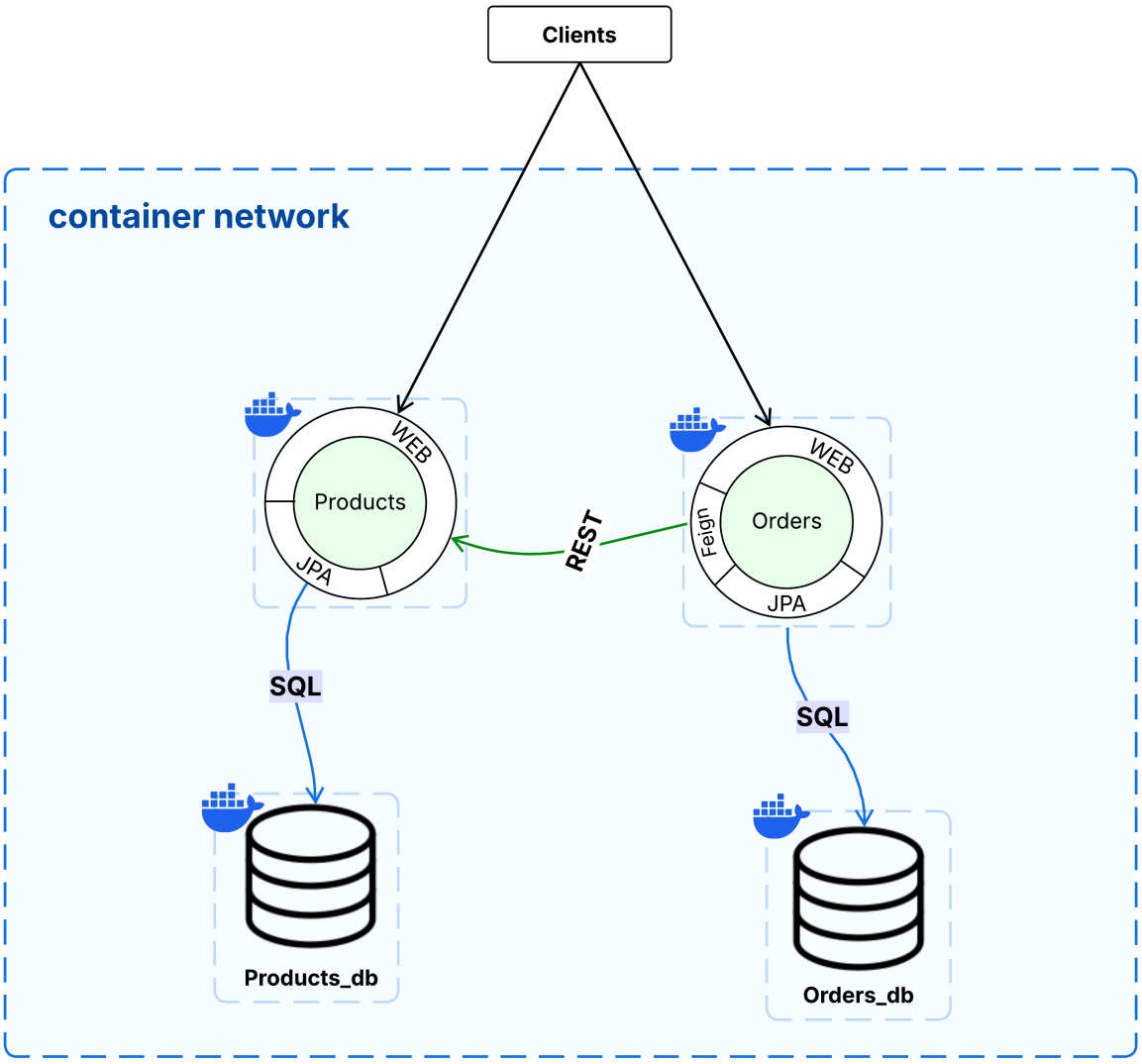

FIGURE 2 – Architecture Microservice sans Middleware 1

L'application a été décomposée en deux services principaux éxposant plusieurs endpoints REST :

1.2.1 Service des Produits (Products Service)

Ce service est responsable de la gestion du catalogue des produits. permettant de :

— Consulter la liste des produits disponibles.

— Récuperer un produit par ID.

1.2.2 Service des Commandes (Orders Service)

Ce service gère le cycle de vie des commandes. Il permet aux utilisateurs de :

— Soumettre de nouvelles commandes.

— Consulter l'historique de leurs commandes.

1.2.3 Communication Inter-services

Il existe une dépendance critique : lorsqu'un utilisateur soumet une commande, le *Service des Commandes* doit **nécessairement** contacter le *Service des Produits* **(communication synchrone)** afin de vérifier la disponibilité du stock avant de valider la transaction.

2 Activité Pratique 1

Dans cette première activité, nous allons observer le comportement de notre système distribué lorsqu'il ne dispose pas d'aucun Middleware.

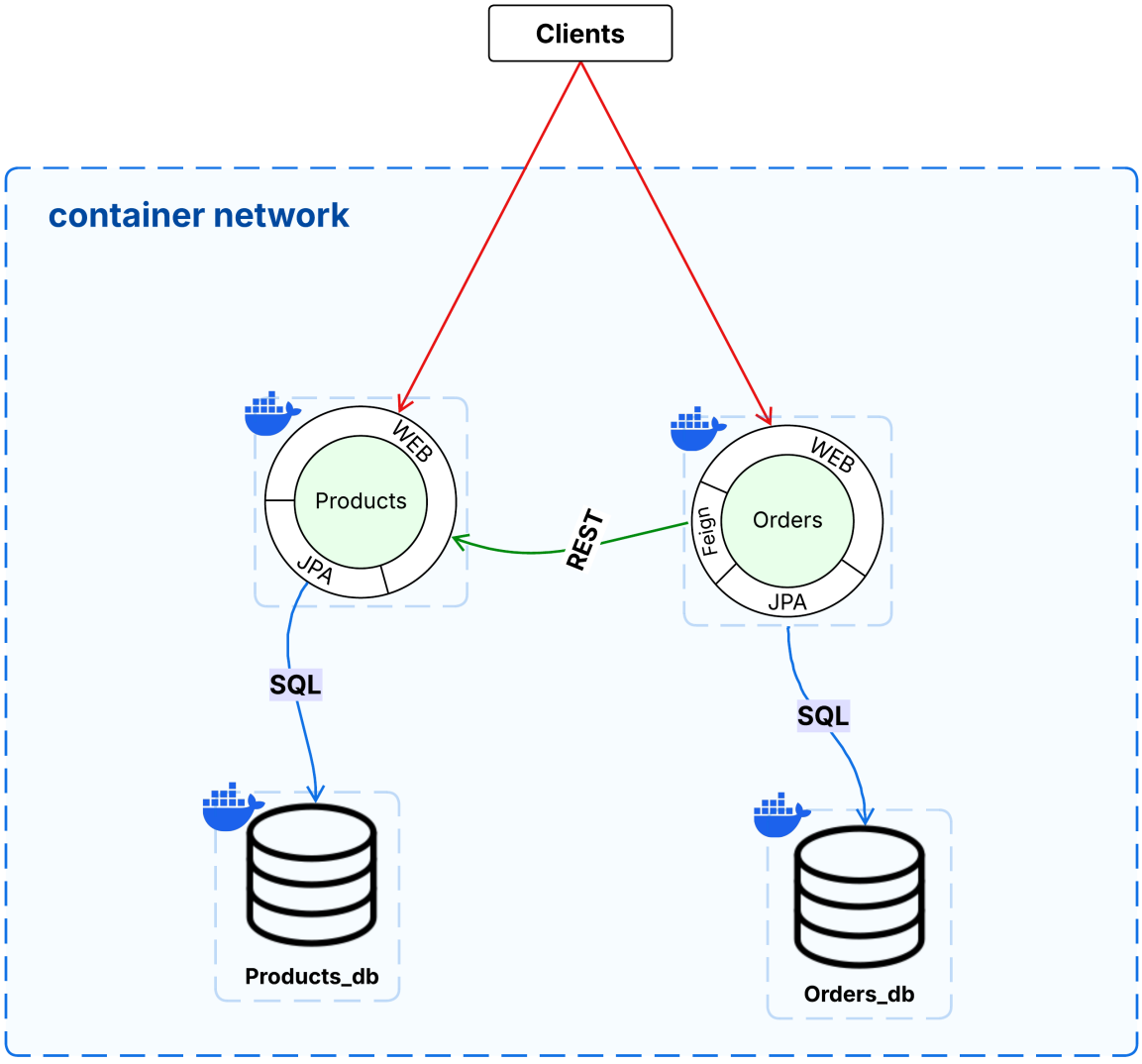

FIGURE 3 – Architecture Microservice sans Middleware 2

2.1 Tâche 1 : Démarrage de l'Architecture

Nous allons commencer par lancer tous les microservices de la première version de l'architecture.

★ Info Docker

Les services **tournent dans des conteneurs isolés**. on a **configuré Docker** (`compose.yaml`) pour relier("mapper") certains ports de notre machine hôte vers les conteneurs, afin de les rendre accessibles (ex : **Port Machine 8081 → Port Conteneur 8080**).

Commande de Lancement

Ouvrez un terminal dans `microservices-without-a-gateway` et exécutez :

```
docker-compose up
```

Vérification des Conteneurs

Ouvrez un terminal et exécutez :

```
docker container ls
```

Une fois lancés, nous observons que nos services sont accessibles sur des ports différents :

— **Orders Service** : `localhost:8081`

— **Products Service** : `localhost:?`

2.2 Tâche 2 : Le Labyrinthe des Adresses (Limitation 1)

L'objectif de cette tâche est de vous mettre à la place d'un client (Frontend ou Mobile) qui doit consommer nos microservices.

Action : Accès aux Services

Ouvrez votre navigateur (ou Postman) et tentez d'accéder aux données :

1. Récupérer la liste des produits : `http://localhost:?/api/products`
2. Récupérer la liste des commandes : `http://localhost:8081/api/orders`

Observation : Pour passer d'un service à l'autre, vous avez dû manuellement modifier le **numéro de port** (? ↔ 8081) dans l'URL.

Limitation Identifiée

Chaque client est forcé de **connaître l'adresse exacte de chaque service backend (IPs et Ports)** et si la topologie réseau change (migration serveur, changement d'IP), le client doit être mis à jour, et il doit maintenir un registre d'addresses.

★ Local vs Production

Dans notre environnement de Lab (Docker Desktop), cette limitation semble mineure car tout est sur `localhost` et seule le port change.
Cependant, dans un environnement de **Production réel** (Cluster Kubernetes), chaque service tournerait sur une machine ou une **IP différente** (ex : `192.168.1.10` pour les produits, `192.168.1.20` pour les commandes). *(en fait dand les environements Cloud chaque conteneur a une addresse IP addressable depuis l'extérieur)*

2.3 Tâche 3 : Le Load Balancing Impossible (Limitation 2)

Nous allons simuler une montée en charge (ex : Black Friday) en **ajoutant des instances supplémentaire du service Produits**.

Action : Scaling Horizontal

Ouvrez un terminal et ajoutez deux instances puis Vérifiez les Conteneurs :

```
docker-compose up -d --scale products-service=3
```

```
docker container ls
```

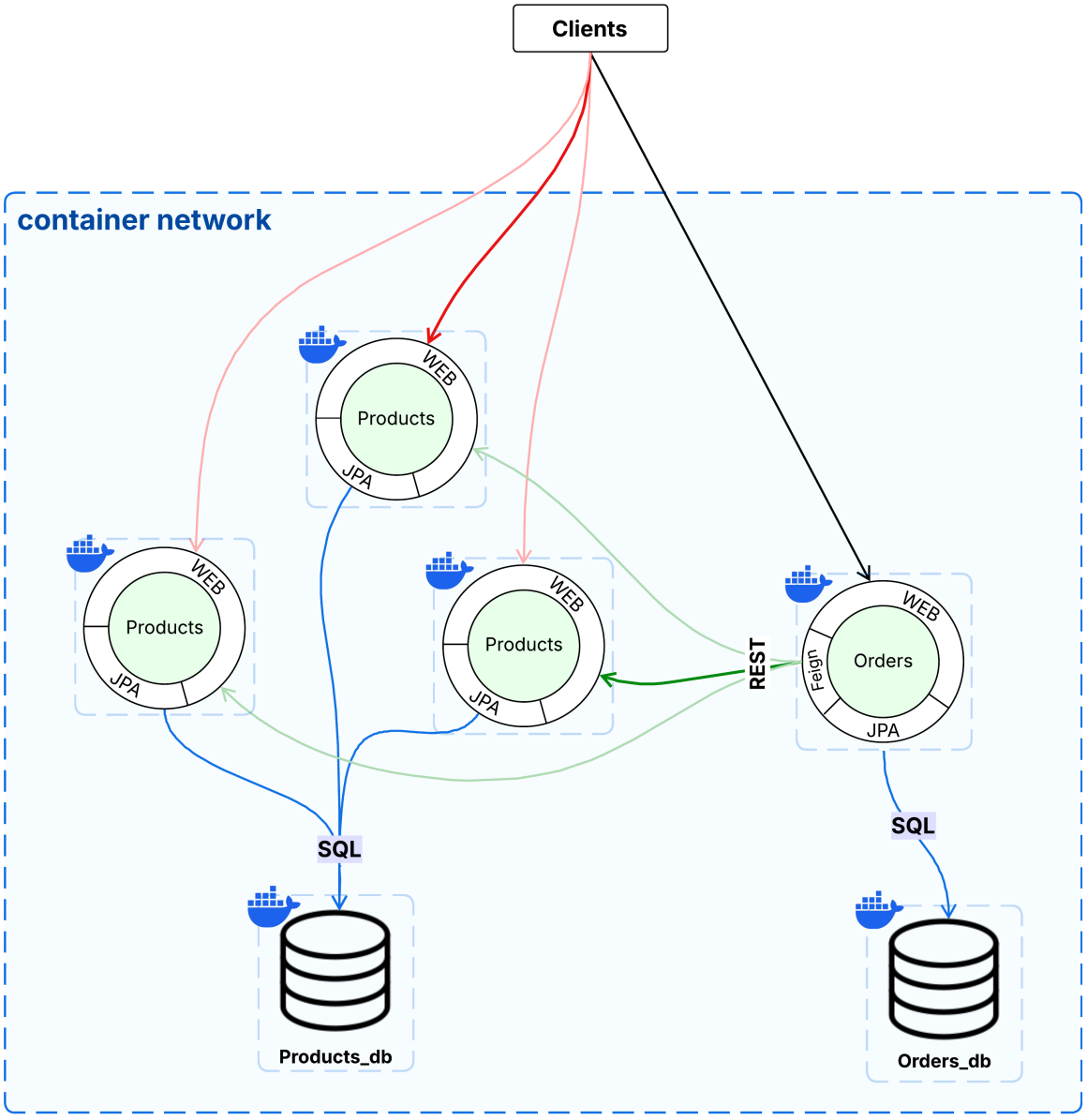

FIGURE 4 – Architecture Microservice sans Middleware après le *"Scaling"*

Analyse Critique : Nous avons maintenant 3 instances de produits. Posez-vous les deux questions fondamentales suivantes :

Question 1 : Accès Externe

Comment le Client (Utilisateur Web) peut-il répartir sa charge ?
Réponse : Il ne peut pas. Il devrait récuperer tous les addresses des services et implementé sont propre algorithme de load balancing..... C'est presque impossible.

Question 2 : Communication Inter-Services

Comment le Service Commandes sait-il quelle instance appeler ?
Réponse : Il ne sait pas. Dans son code, l'URL est fixée. Il continuera d'appeler la première instance du service des produits, ignorant totalement les deux nouvelles. Le scaling est donc **inutile** sans *mécanisme de découverte (Registry)* ou *de répartition (Load Balancer)*.

Limitation Identifiée

Le système n'est pas "élastique", Les Clients ne sont capable de répartir la charge, Ajouter des instances ne sert a rien.

2.4 Tâche 4 : L'Instabilité des Adresses (Limitation 3 - IPs Éphémères)

Nous allons prouver que l'adresse IP d'un service n'est pas fiable dans le temps.

Étape 1 : Relever l'IP Initiale

Inspectez le conteneur 'products-service-1' pour obtenir son IP actuelle :

```
docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' microservices-without-a-gateway-products-service-2
```

Notez cette IP (ex : 172.18.0.3).

Étape 2 : Simulation de Panne et Remplacement

Arrêtons, supprimons, puis recréons le service :

```
docker stop microservices-without-a-gateway-products-service-2
```

```
docker rm microservices-without-a-gateway-products-service-2
```

```
docker-compose up -d products-service
```

Étape 3 : Relever la Nouvelle IP

Inspectez le nouveau conteneur créé :

```
docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' microservices-without-a-gateway-products-service-2
```

Vous constaterez que la nouvelle IP est ***différente*** de la première.

Limitation Identifiée

L'identité réseau (IP) d'un conteneur est volatile. dans cette architecture le Service Commandes code en dure l'IP et le port d'un ancien conteneur qui roule le service produits.

2.5 Tâche 5 : La Duplication des Efforts (Limitation 4 - Cross-Cutting Concerns)

Enfin, observons comment sont gérés les aspects transversaux comme la sécurité.

Simulation : Authentification Stateless

Nous **simulons ici un mécanisme d'authentification Stateless** (sans session). Cela signifie que chaque microservice ne fait confiance à personne par défaut. Il doit donc systématiquement valider la signature cryptographique du Token (JWT) de chaque requête qu'il reçoit, même si elle vient d'un autre service interne.

Action : Audit des Logs

1. Ouvrez deux terminaux séparés pour suivre les logs en temps réel :

```
# Terminal 1
docker-compose logs -f orders-service
```

```
# Terminal 2
docker-compose logs -f products-service
```

2. Effectuez une seule requête "Créer une Commande" (via Postman sur le port 8081).

Observation : Regardez simultanément les deux terminaux. Vous verrez apparaître le message suivant **deux fois** :

```
[SECURITY] Verifying Token signature...
```

Limitation Identifiée

le code de sécurité est **dupliqué** dans chaque Microservices, Ce qui rend les services surchargé de responsabilité au dehors de leurs logique métier.
Pour une seule action métier de l'utilisateur, nous avons payé le coût de la vérification de sécurité **deux fois** (une fois à l'entrée de *Commandes*, une fois à l'entrée de *Produits*).

2.6 Conclusion

Bilan Négatif

Dans cette architecture, le Client (Frontend/Mobile) :

1. le Client **Dépend de l'implémentation interne** du système distribué (IPs, Ports) au lieu de dépendre d'une interface abstraite (Facade).
2. le Client **Est surchargé de responsabilités** (Service Discovery, Load Balancing), violant le *Single Responsibility Principle*. Normalement, il devrait se contenter d'afficher des données et envoyer des requete simple.
3. Les aspects transversaux comme l'Authentification et les Logs sont supliquées dans chaque microservice.

3 Transition vers l'Architecture Régulée

Avant de passer à la solution, nous devons arrêter l'architecture actuelle pour libérer les ressources et les ports.

Nettoyage

```
docker-compose down
cd ..
```

4 Activité Pratique 2

Pour résoudre les limitations observées, nous introduisons un nouveau composant : le Middleware **API Gateway**. Celui-ci va agir comme un **Point d'Entrée Unique (Single Entry Point)** pour tous les clients.

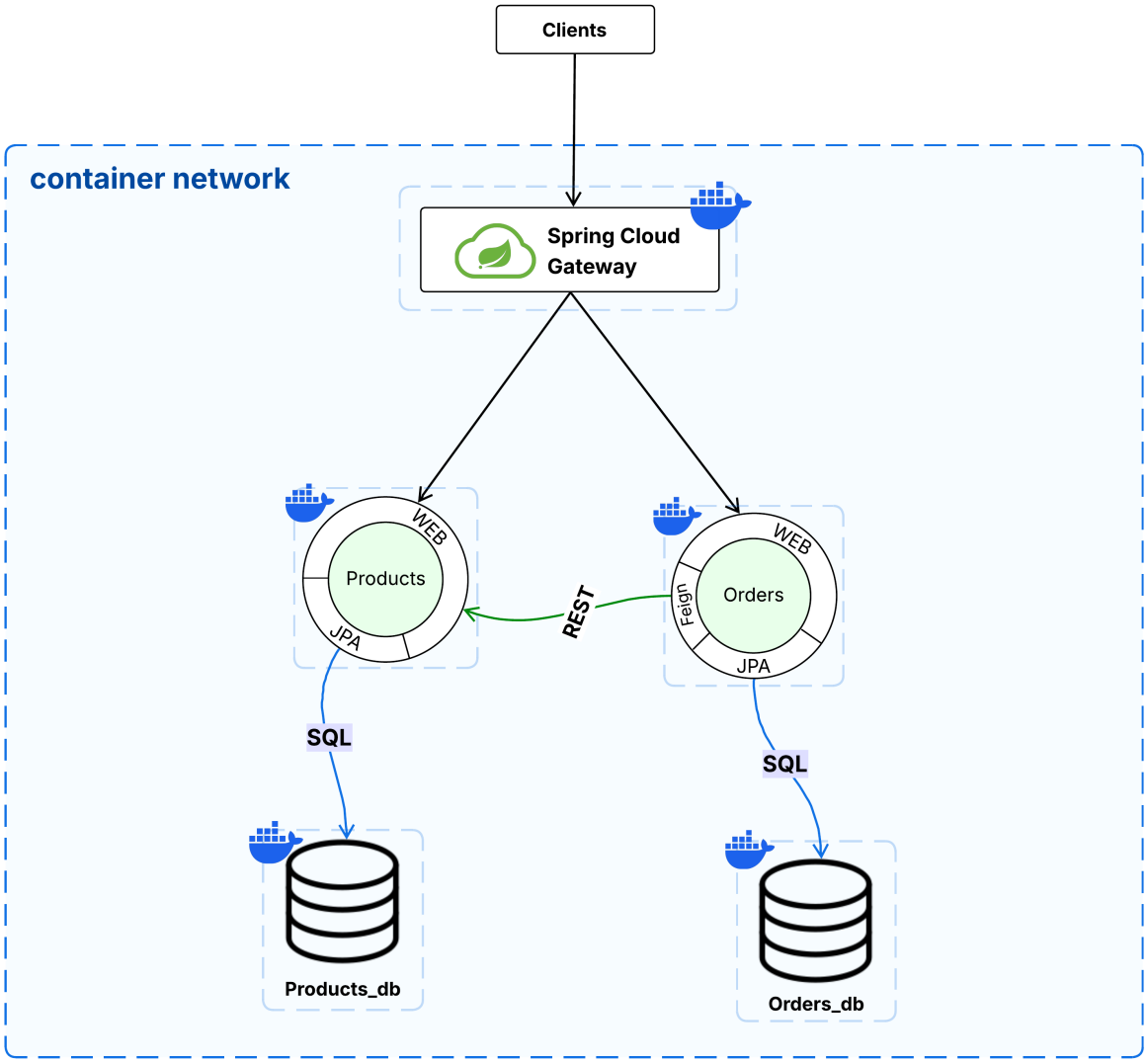

FIGURE 5 – Architecture Microservice avec Middleware (API Gateway)

4.1 Tâche 1 : Mise en Place et Point d'Entrée Unique

Lancement

Placez-vous dans le dossier `microservices-with-a-gateway` et lancez :

```
docker-compose up
```

Vérification des Conteneurs

Ouvrez un terminal et exécutez :

```
docker container ls
```

Une fois lancés, vérifions l'accès à nos services :

- **API Gateway** : Accessible via `localhost:8888`
- **Orders Service** : **Inaccessible directement**
- **Products Service** : **Inaccessible directement**

★ Sécurité par Isolation

Dans cette architecture, le client n'a **aucun accès direct** aux services backend.

1. Ils ne sont pas exposés sur la machine hôte.
2. Ce sont des processus isolés dans le réseau privé Docker.
3. Seule la **Gateway** est exposé vers l'extérieur depuis le Port 8888 et joue le role d'une "porte" vers le réseau interne ou les autres microservices sont déployés.

Le client est donc **obligé** de passer par la Gateway, ce qui nous permet de contrôler et sécuriser tout le trafic entrant.

4.2 Tâche 2 : Test du Point d'Entrée Unique

Action : Accès aux Services depuis le Middleware

Ouvrez votre navigateur (ou Postman) et tentez d'accéder aux données :

1. Récupérer la liste des produits : `http://localhost:8888/api/products`
2. Récupérer la liste des commandes : `http://localhost:8888/api/orders`

Solution : Abstraction

Le client ne connaît plus qu'une seule adresse (Port 8888). La Gateway route les requêtes vers les bons services en interne. Le client ne dépend plus de l'implémentation interne du système :

— Il n'a plus besoin de connaître **les adresses exactes (IPs et Ports)** de chaque service backend.

— Il n'a plus besoin de **maintenir un registre d'adresses** ou d'être mis à jour si la topologie réseau change.

Tout passe désormais par un point unique.

4.3 Tâche 3 : Load Balancing et Scalabilité Transparente

Simulant une montée en charge en **ajoutant 2 instances du service Produits**.

Action : Scaling Horizontal

Ouvrez un terminal et ajoutez deux instances puis Vérifiez les Conteneurs :

```
docker-compose up -d --scale products-service=2
docker container ls
```

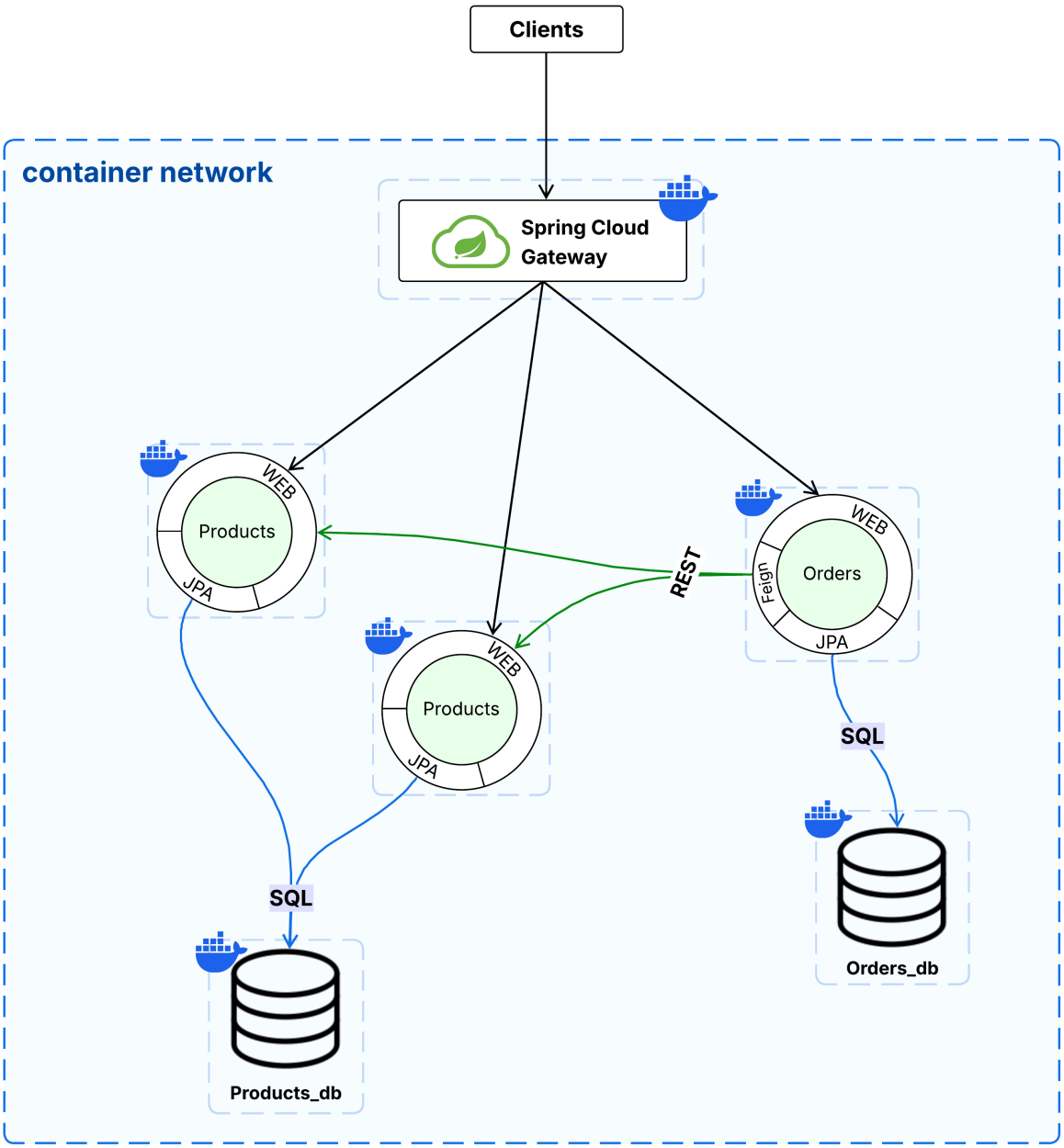

FIGURE 6 – Architecture Microservice avec Middleware (API Gateway) après le *"Scaling"*

Test de Répartition de Charge

1. Dans un terminal, surveillez les logs de **toutes les instances** de produits :

```
docker-compose logs -f products-service
```

2. Envoyez **5 ou 6 requêtes** successives à la Gateway : *(Utilisez votre navigateur en rafraîchissant la page http://localhost:8888/api/products)*

Observation : Regardez les logs. Vous verrez que les requêtes sont traitées alternativement par l'instance 1, puis l'instance 2, etc.

Solution : Scalabilité Transparente

Le client continue d'appeler la même adresse unique (`localhost:8888`). Cependant, en coulisses, la Gateway (aidée par le réseau Docker) répartit automatiquement la charge sur les nouvelles instances disponibles. L'ajout de serveurs est devenu **transparent** pour le client.

4.4 Tâche 4 : Centralisation des Cross-Cutting Concerns

Au lieu de dupliquer le code de sécurité ou de logging dans chaque microservice (Produits, Commandes, Paiement...), nous pouvons désormais le **centraliser** au niveau de la Gateway.

Solution : Centralisation

La Gateway intercepte 100% du trafic entrant. C'est donc l'endroit idéal pour placer des **Filtres Globaux** :

- **Authentification :** Vérifier le Token JWT *avant* même que la requête n'atteigne les services.
- **Rate Limiting :** Bloquer les abus (ex : max 10 requêtes/seconde).
- **Logging :** Enregistrer qui fait quoi.

5 Conclusion Générale

Cette activité pratique vous a permis de **vivre concrètement** le passage d'une architecture microservices non régulée vers une architecture **régulée par un middleware**, en l'occurrence une **API Gateway**.
Le message clé de ce TP est le suivant : dans un système distribué réel, les middlewares ne sont pas un luxe mais une **nécessité architecturale**. Ils permettent :

— de **casser le couplage** entre clients et microservices ;

— de **gérer la complexité** liée au réseau (découverte, scalabilité, résilience) ;

— de **simplifier le code métier** en externalisant les préoccupations transversales.

Enfin, ce TP n'est qu'un **premier pas** : l'API Gateway n'est qu'un type de middleware parmi d'autres (ESB, Message Bus, Service Mesh...). L'objectif est que vous puissiez désormais **relier la théorie de l'atelier** (*Topologies de Middleware d'une entreprise engagée*) à une **expérience pratique concrète**, et comprendre comment un choix d'architecture middleware impacte directement l'**expérience des développeurs, des clients et la qualité globale du système**.