

### III : Les patterns de conceptions des micro-services

### 3 - Les patterns de conception des micro-services

- Les patterns de conception sont des solutions éprouvées à des problèmes courants.
- Dans le contexte des microservices, ces patterns peuvent aider à gérer la complexité inhérente à la construction et à l'exploitation de systèmes distribués.
- Ils organisent l'intelligence de conception dans un format standard et facile à référencer.
- Ils sont le plus souvent flexibles et optionnels.

## **3 - Les patterns de conceptions des micro-services**

**Les patterns  
d'intégration**

**Les patterns de  
gestion de données**

**Le pattern de  
fiabilité**

## 2 - 9 - 1- Les patterns d'intégration

1. Le Pattern composition d'API.
2. API Gateway Pattern (Le pattern de passerelle API).
3. Le Pattern BFF
4. Le Pattern Messagerie Asynchrone.

## 2 - 9 - 1- Le Pattern composition d'API.

### Contexte :

- On a une application à base de Microservices avec une base des données pour chaque service. Par conséquent, il n'est plus facile d'implémenter des requêtes qui combinent les données provenant de plusieurs services.

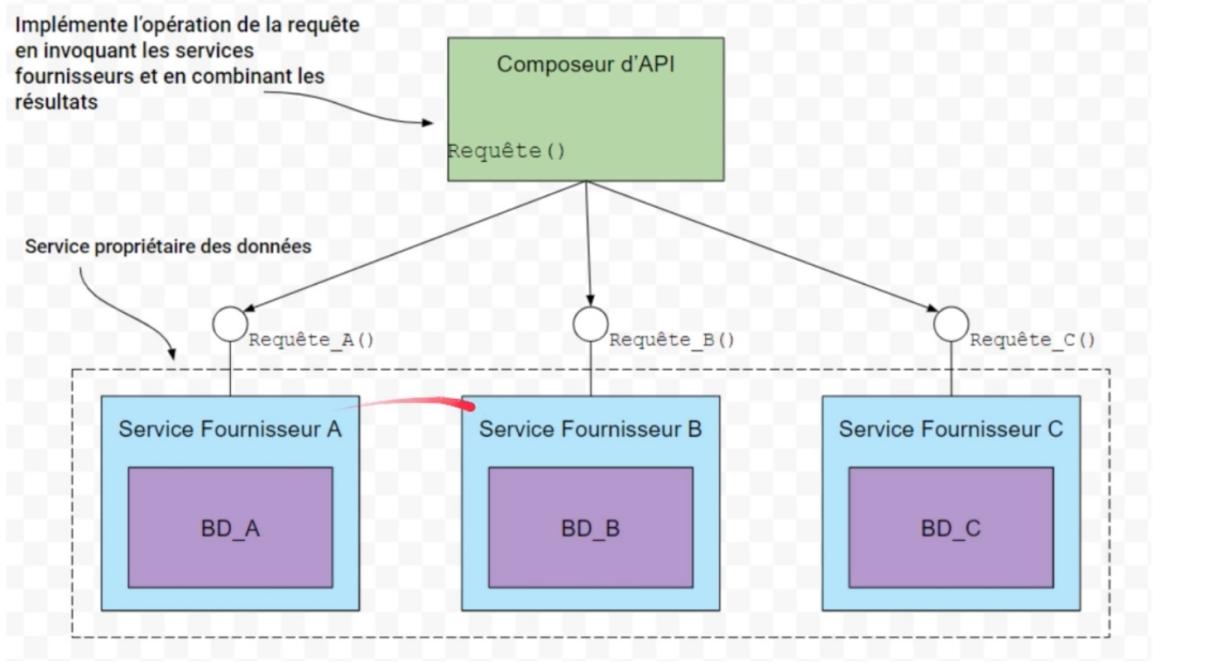
### Problème :

- Comment implémenter des requêtes qui combinent des données provenant de plusieurs microservices ayant chacun sa base de données ?

### Solution

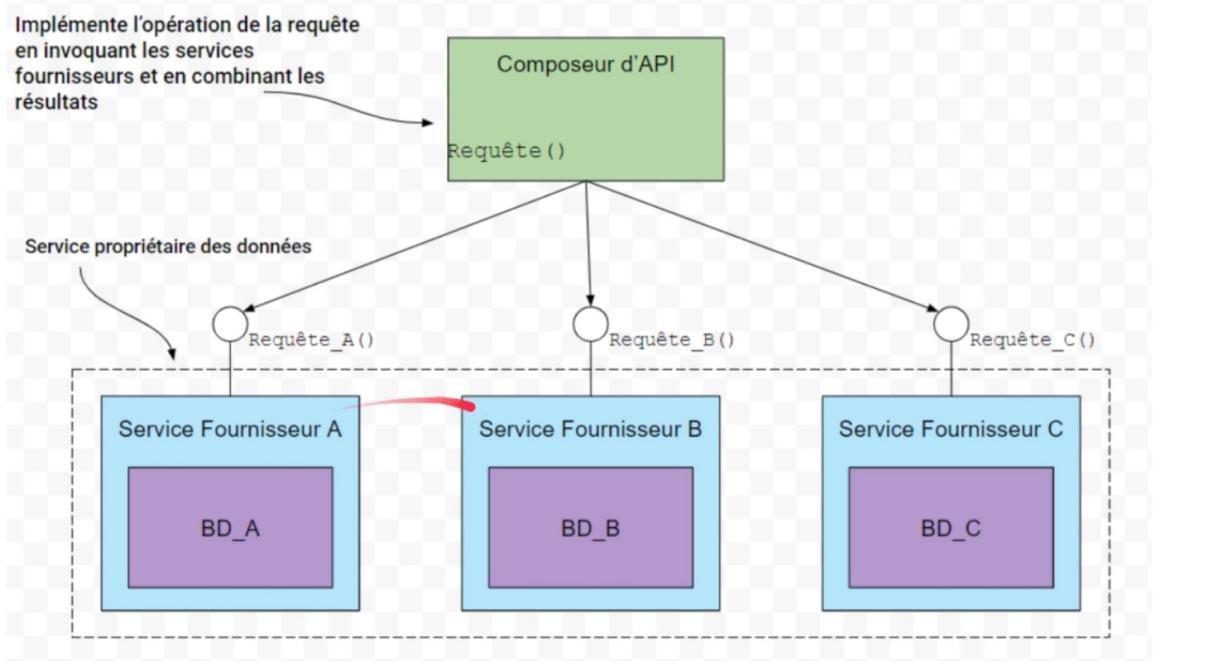
- Utiliser un 'composeur d'API' pour implémenter une telle requête.

## 2 - 9 - 1- Le Pattern composition d'API



1. **Composeur d'API** : Ce composant agit comme un intermédiaire qui reçoit les requêtes externes. Son rôle est d'orchestrer les appels aux microservices appropriés pour satisfaire une requête, qui peut nécessiter de combiner des données de plusieurs services.
2. **Requête ()** : Il s'agit de la fonction exposée par le Composeur d'API que le client peut invoquer. Cette fonction est responsable de l'orchestration des appels aux services nécessaires pour composer la réponse attendue par le client.
3. **Service Fournisseur A/B/C** : Ce sont les microservices qui gèrent une partie spécifique de la logique métier et possèdent leurs propres bases de données (BD\_A, BD\_B, BD\_C). Ils sont responsables de traiter les requêtes qui leur sont spécifiques et de renvoyer des données.

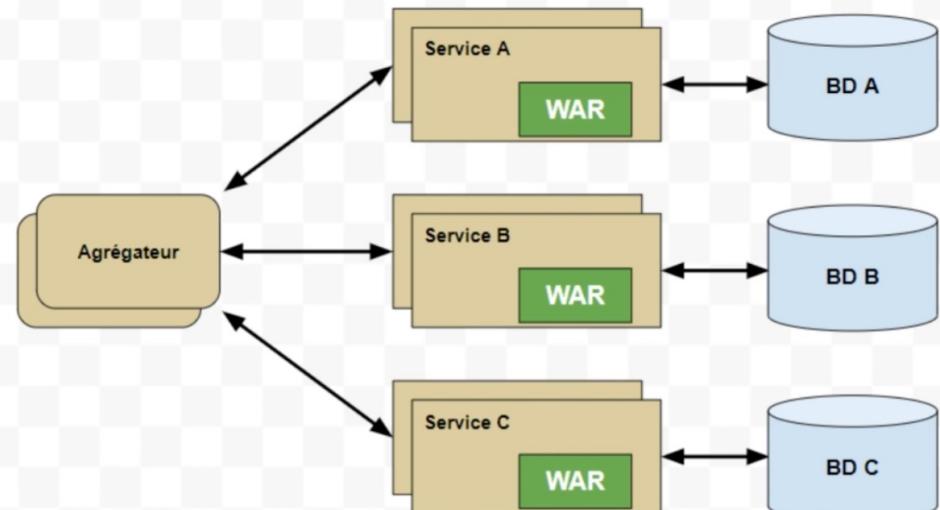
## 2 - 9 - 1- Le Pattern composition d'API



4. **Requête\_A(), Requête\_B(), Requête\_C()**: Ce sont les appels spécifiques que le Composeur d'API fait à chaque microservice. Chaque service a sa propre fonction pour traiter la requête qui lui est envoyée.
5. **Service propriétaire des données** : C'est une mention qui indique que chaque microservice gère et possède sa propre base de données, ce qui est typique dans une architecture de microservices où chaque service est autonome et indépendant des autres services.

## 2 - 9 - 1- Le Pattern d'Aggregateur

### Le pattern Agrégateur

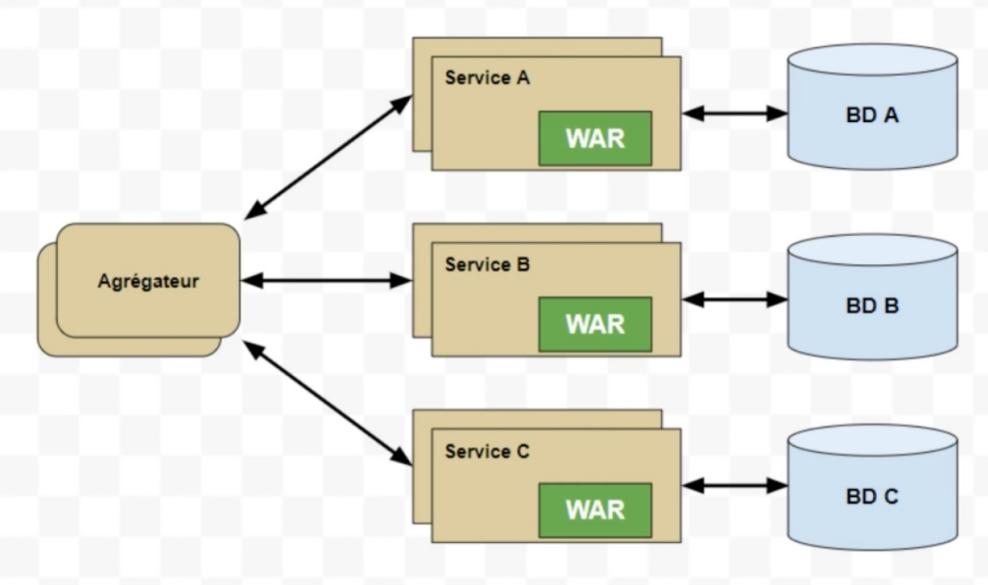


Le pattern **Aggrégateur** fonctionne comme un intermédiaire qui reçoit une demande client, fait des appels à différents services (dans ce cas, Service A, B, et C), chacun potentiellement avec sa propre base de données (BD A, BD B, et BD C), et agrège les résultats pour fournir une réponse unifiée au client.

Proche du pattern par composition d'API.

## 2 - 9 - 1- Le Pattern d'Aggregateur

### Le pattern Agrégateur

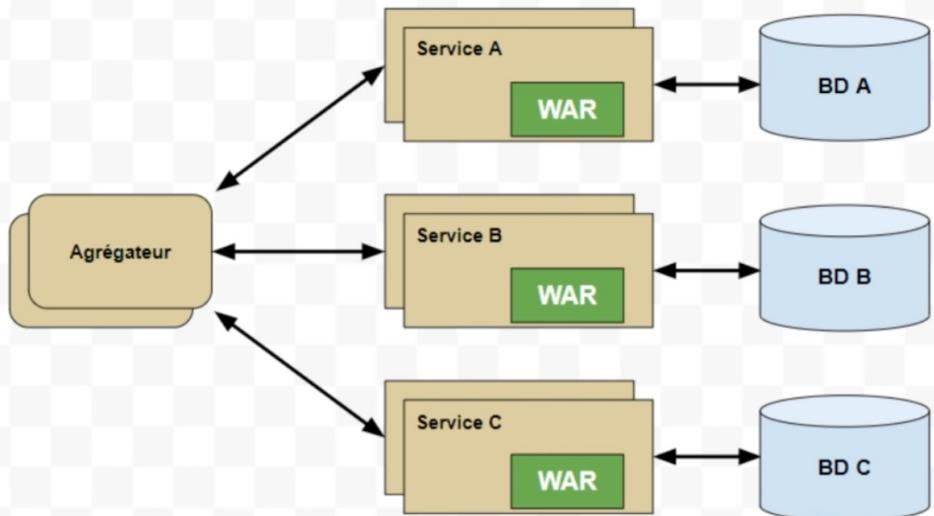


Proche du pattern par composition d'API.

1. **L'Aggrégateur** : Ce composant central est responsable de l'envoi des requêtes aux services appropriés, et de l'agrégation des réponses de ces services en un seul ensemble de données cohérent.
2. **Services (A, B, C)** : Ces blocs représentent différents microservices, chacun possiblement déployé comme un fichier WAR (Web Application Archive), qui est un format couramment utilisé pour la distribution de collections de fichiers JavaServer Pages, de servlets Java, de classes Java, de fichiers XML, et d'autres ressources qui ensemble constituent des applications web.
3. **Bases de Données (BD A, B, C)** : Chaque service a sa propre base de données, ce qui suggère que chaque service fonctionne de manière indépendante et peut-être en isolation, une caractéristique commune dans les architectures de microservices.

## 2 - 9 - 1- Le Pattern d'Aggregateur

### Le pattern Agrégateur



Proche du pattern par composition d'API.

*La différence principale entre ce pattern Aggrégateur et celui de la Composition d'API est la façon dont ils gèrent l'intégration des réponses des différents services :*

**Pattern Aggrégateur** : Ce pattern met l'accent sur l'agrégation simple des réponses des services sans une logique métier complexe. C'est un point central où toutes les réponses des services sont simplement combinées et renvoyées au client.

**Composition d'API** : Avec la Composition d'API, il y a un niveau supplémentaire de traitement, où la logique métier peut être implémentée pour non seulement agréger des données mais aussi les transformer, les combiner de façon complexe, ou effectuer des opérations conditionnelles sur elles avant de renvoyer la réponse.

## 2 - 9 - 1- Le Pattern composition d'API.

- **Avantage :**
  - C'est un moyen simple d'interroger des données dans une architecture de microservices
- **Inconvénient :**
  - Certaines requêtes entraîneraient des jointures en mémoire inefficaces pour des gros volumes de données.

## 2 - 9 - 1- Le Pattern API Gateway.

### Problème

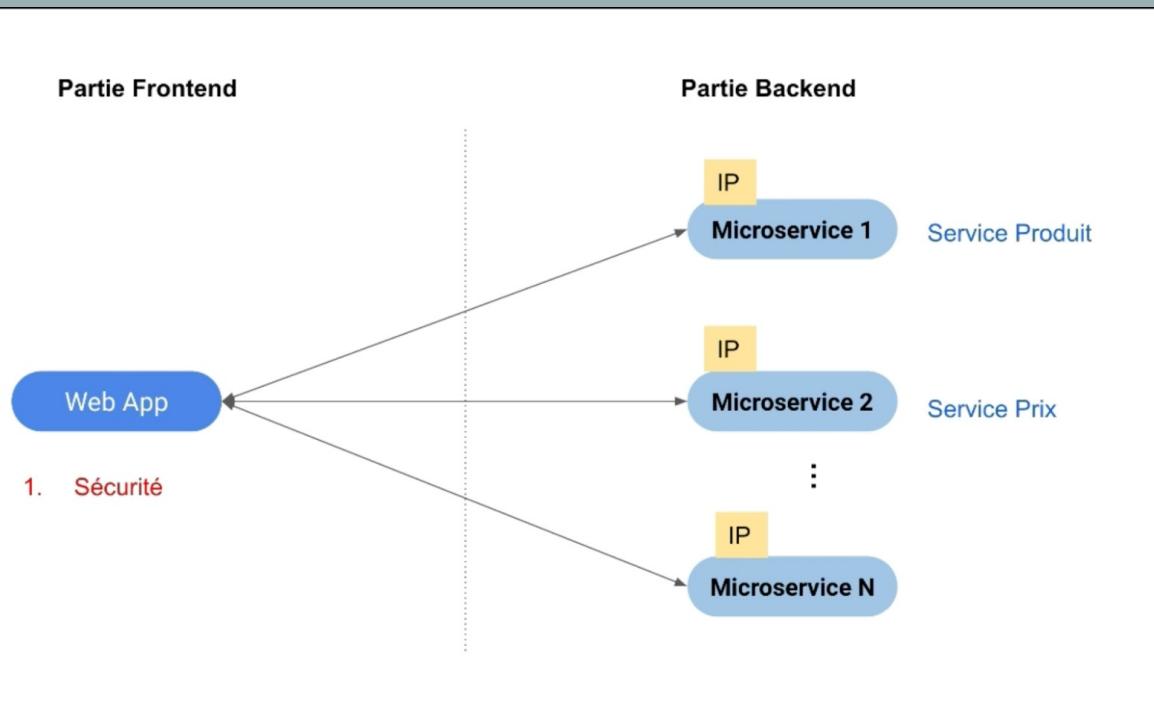
- Lorsqu'une application est divisée en microservices plus petits, quelques problèmes doivent être résolus :
  - Sur différents canaux (comme les ordinateurs de bureau, les mobiles et les tablettes), les applications ont besoin de données différentes pour répondre au même service backend, car l'interface utilisateur peut être différente.
  - Différents consommateurs peuvent avoir besoin d'un format différent pour les réponses des microservices réutilisables. Qui effectuera la transformation des données ou la manipulation des champs ?
  - Comment gérer différents types de protocoles, dont certains peuvent ne pas être pris en charge par le microservice du producteur.

## 2 - 9 - 1- Le Pattern API Gateway.

### Solution

- Une passerelle API permet de répondre à de nombreuses préoccupations soulevées par la mise en œuvre des microservices, sans se limiter à celles vu précédemment.
- Une passerelle API est le point d'entrée unique pour tout appel de microservice.
- Il peut fonctionner comme un service proxy pour acheminer une requête vers le microservice concerné, en extrayant les détails du producteur.
- Il peut diffuser une demande vers plusieurs services et regrouper les résultats à renvoyer au consommateur.
- Les API universelles ne peuvent pas répondre à toutes les exigences des consommateurs ; cette solution peut créer une API fine pour chaque type spécifique de client.
- Il peut également convertir la requête de protocole (par exemple AMQP) en un autre protocole (par exemple HTTP) et vice versa afin que le producteur et le consommateur puissent la gérer.
- Cela peut également décharger la responsabilité d'authentification/autorisation du microservice.

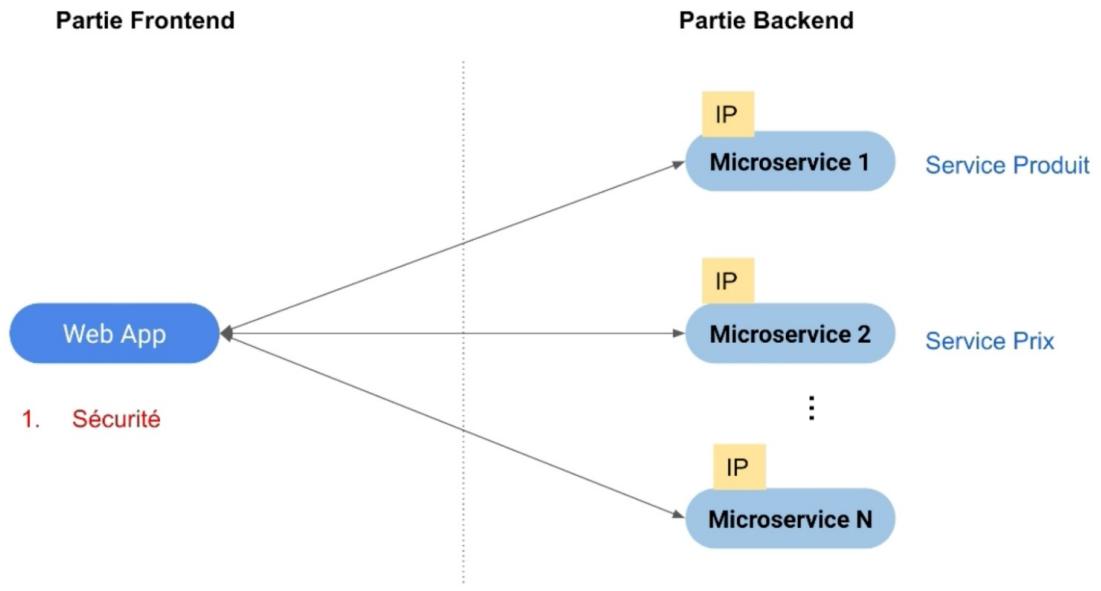
## 2 - 9 - 1- Le Pattern d'API Gateway



### Les limites sans l'API Gateway

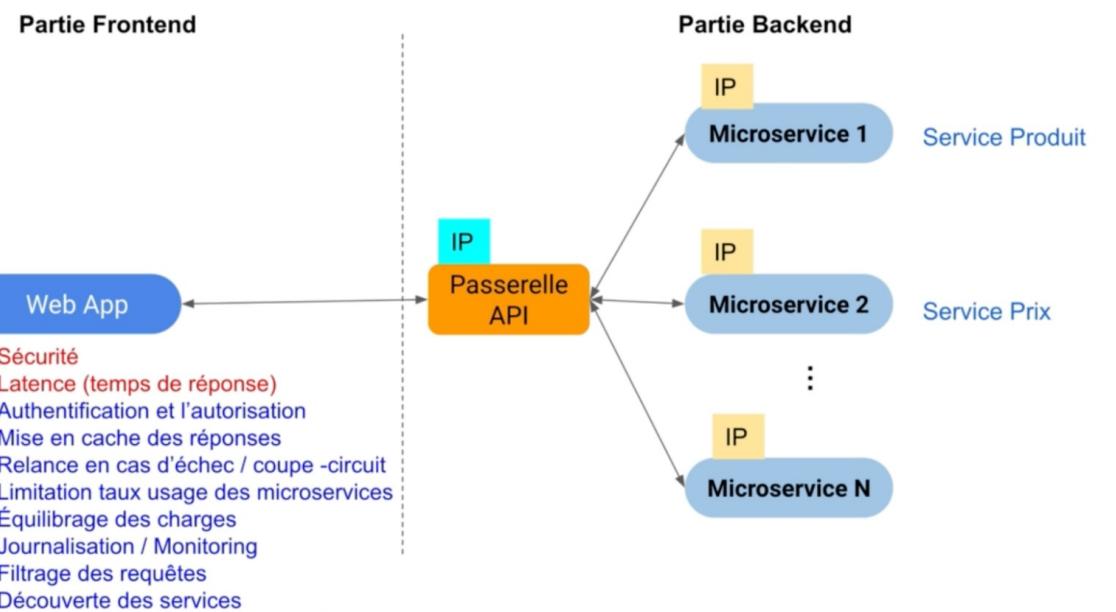
- Sécurité** : Chaque microservice doit implémenter sa propre sécurité, ce qui peut entraîner des incohérences et des vulnérabilités. Sans une API Gateway qui agit comme un point d'accès uniifié, il est difficile d'appliquer des politiques de sécurité cohérentes et robustes.
- Cross-cutting concerns** : Les préoccupations transversales telles que la journalisation, la surveillance, le rate limiting, et l'authentification doivent être gérées par chaque microservice de façon individuelle, ce qui peut mener à une duplication d'efforts et un manque d'uniformité.
- Scalabilité** : Sans une API Gateway, il peut être difficile de gérer la scalabilité de manière efficace. La gateway peut offrir un mécanisme de load balancing pour distribuer la charge entre les instances des microservices.

## 2 - 9 - 1- Le Pattern d'API Gateway



5. **Complexité pour le client :** Le client (dans ce cas, la web app) doit connaître tous les points de terminaison et les protocoles de communication pour chaque microservice, ce qui augmente la complexité du code client et sa maintenance.
6. **Résilience :** En l'absence d'une API Gateway, la résilience du système peut être compromise. La gateway peut implémenter des patterns comme les circuit breakers pour prévenir les défaillances en cascade.
7. **Performance :** La web app pourrait avoir besoin de faire plusieurs appels réseau pour construire une réponse complète, ce qui peut être inefficace et augmenter la latence.
8. **Agilité :** Mettre à jour les points de terminaison ou les protocoles de communication nécessitera des modifications dans la web app, ce qui réduit l'agilité et augmente le couplage entre le frontend et le backend.

## 2 - 9 - 1- Le Pattern d'API Gateway

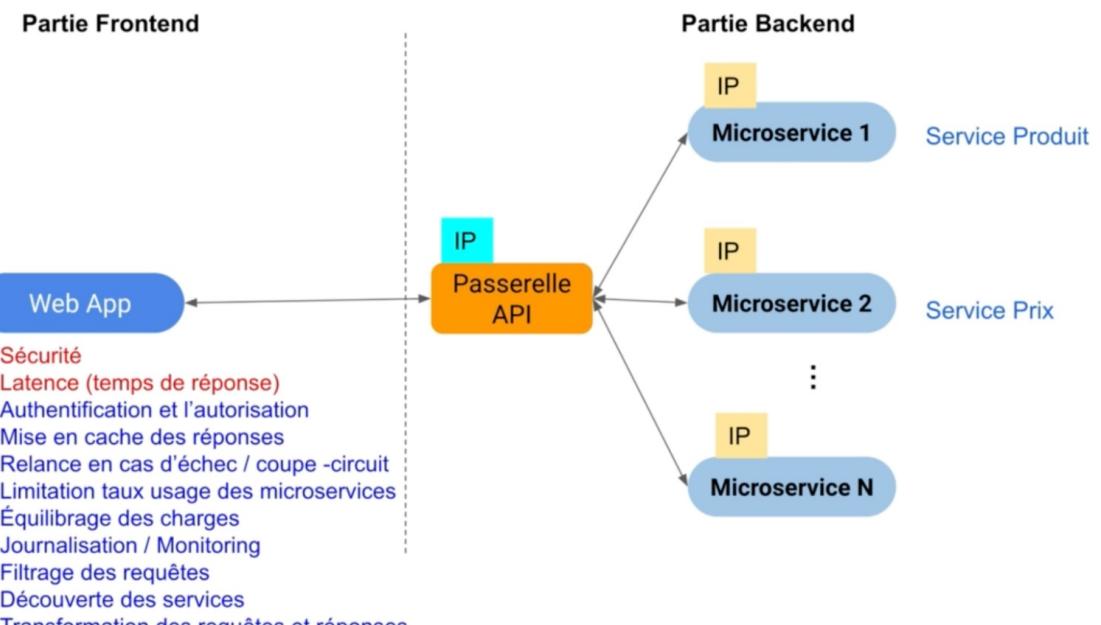


La Passerelle API est positionnée entre la Web App et les microservices et sert de point d'entrée unique pour les requêtes allant du Frontend vers le Backend.

Cela offre plusieurs avantages, comme indiqué par la liste à gauche de l'illustration :

- 1. Sécurité** : La passerelle peut gérer les aspects de sécurité, comme l'authentification et l'autorisation, avant de transmettre les requêtes aux microservices.
- 2. Latence (temps de réponse)** : Elle peut améliorer les performances en réduisant la latence grâce à des mécanismes comme la mise en cache des réponses.
- 3. Authentification et autorisation** : La passerelle vérifie l'identité de l'utilisateur et ses droits d'accès avant de permettre l'accès aux microservices.
- 4. Mise en cache des réponses** : Pour accroître l'efficacité et réduire la charge sur les microservices, la passerelle peut stocker temporairement les données fréquemment demandées.

## 2 - 9 - 1- Le Pattern d'API Gateway



5. **Relance en cas d'échec / coupe-circuit :** Elle peut implémenter des stratégies de résilience, comme les "circuit breakers" pour éviter les défaillances en cascade.
6. **Limitation taux usage des microservices (Rate Limiting) :** Pour prévenir la surcharge des services, elle peut limiter le nombre de requêtes qu'un utilisateur peut envoyer dans un intervalle de temps donné.
7. **Équilibrage des charges (Load Balancing) :** Répartition des requêtes entrantes de manière équilibrée entre les différentes instances des microservices.
8. **Journalisation / Monitoring :** La passerelle peut enregistrer les requêtes et les réponses pour le suivi, le diagnostic et l'analyse de sécurité.
9. **Filtrage des requêtes :** Elle peut filtrer les requêtes entrantes pour s'assurer qu'elles sont conformes avant de les transmettre aux microservices.
10. **Découverte des services :** La passerelle peut aider les services à découvrir d'autres services disponibles dans l'architecture.
11. **Transformation des requêtes et réponses :** Elle peut convertir les requêtes et les réponses pour assurer la compatibilité entre les clients et les microservices qui peuvent utiliser différents formats.

## 2 - 9 - 1- Le Pattern API Gateway.

### Les acteurs majeurs de l'espace API Gateway :

- Apigee
- MuleSoft
- Axway
- Kong Gateway
- Young App
- SnapLogic
- Akana API Management
- Oracle API Gateway
- TIBCO Exchange Gateway
- Azure API Gateway
- 3scale
- Ambassador
- Ocelot
- Amazon AWS API Gateway
- Tyk.io

## 2 - 9 - 1- Le Pattern BFF

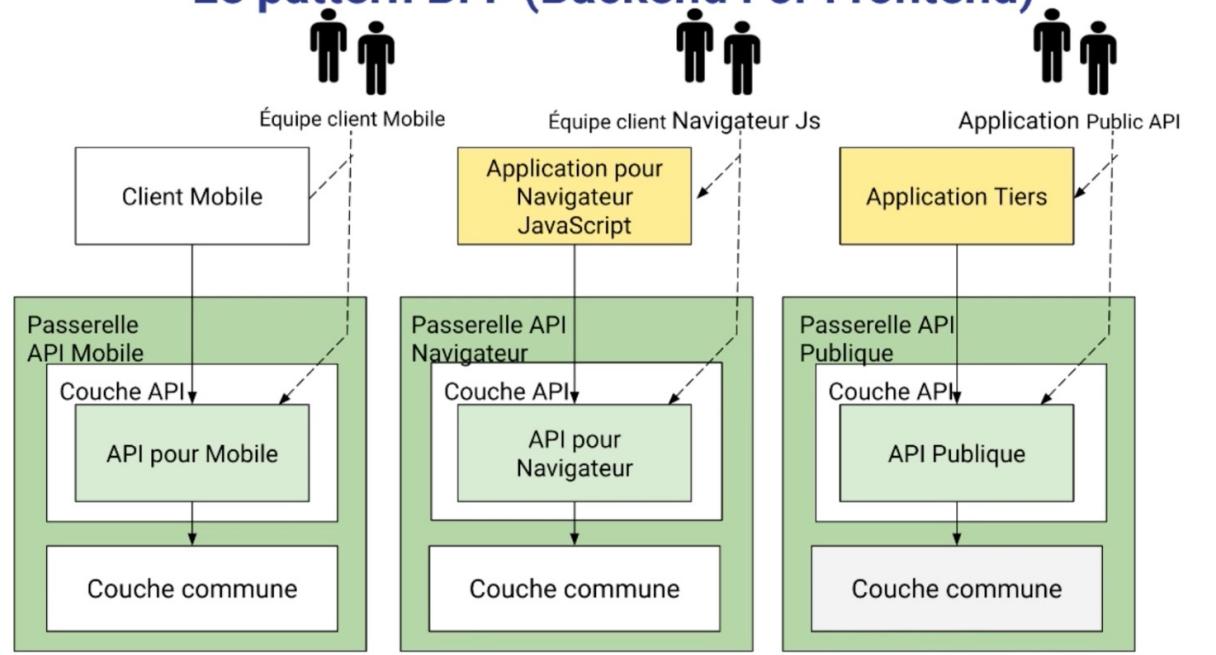
- Le terme "BFF" dans le contexte du développement de logiciels est l'acronyme de "Backend For Frontend".
- C'est un modèle d'architecture dans lequel le backend est spécifiquement conçu pour répondre aux besoins d'un frontend particulier, comme un client web, mobile ou un appareil IoT.
- L'idée est qu'au lieu d'avoir un seul backend monolithique qui expose une API unique pour tous les clients, chaque type de frontend a son propre backend BFF personnalisé.
- Ce backend est optimisé pour les besoins spécifiques de chaque client, ce qui peut inclure des différences dans les données requises, les performances, les opérations CRUD, les protocoles de communication, etc.

## 2 - 9 - 1- Le Pattern BFF

- En utilisant des BFFs, les développeurs peuvent créer des expériences utilisateur plus ciblées et efficaces.
- Ils peuvent réduire la quantité de données transmises sur le réseau, réduire la latence et améliorer la sécurité en exposant uniquement les données et les opérations nécessaires pour chaque client.
- Cependant, cela peut également entraîner une duplication de la logique et des efforts supplémentaires pour maintenir plusieurs backends.
- C'est pourquoi l'utilisation de BFFs doit être équilibrée avec les besoins et les ressources du projet.

## 2 - 9 - 1- Le Pattern BFF

### Le pattern BFF (Backend For Frontend)



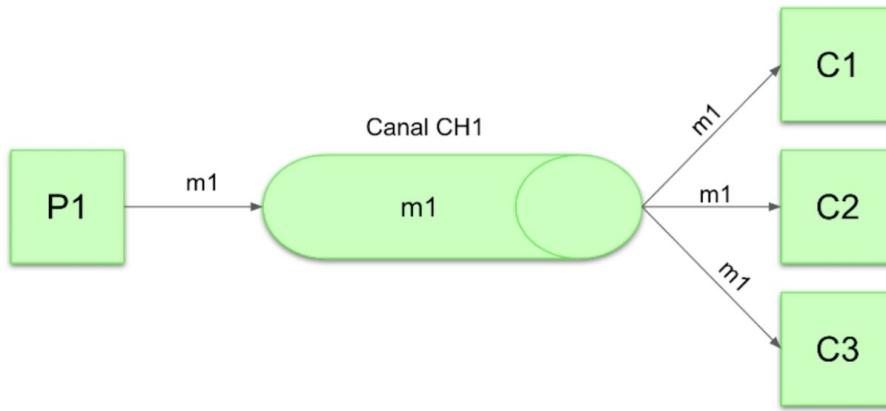
- L'objectif de cette architecture est de permettre une spécialisation des interfaces backend pour mieux répondre aux besoins spécifiques des différents types de clients frontaux, tout en partageant une base commune pour les fonctions qui ne nécessitent pas de spécialisation.
- Cela permet une meilleure performance, personnalisation, et potentiellement une meilleure sécurité et facilité de maintenance.

## 2 - 9 - 1- Le Pattern de messagerie asynchrone

- **Le pattern de messagerie asynchrone** dans le contexte de l'architecture des systèmes d'information est une méthode de communication entre différents services ou composants qui ne nécessite pas que les parties communiquantes soient disponibles en même temps.
- Cela permet à un composant d'envoyer un message sans attendre de réponse immédiate, et le récepteur peut traiter le message et répondre à son propre rythme.

## 2 - 9 - 1- le pattern publication/abonnement (pub/sub).

### Le modèle Publication / Abonnement



- Dans ce modèle, les producteurs de messages (les éditeurs) ne communiquent pas directement avec les consommateurs (les abonnés).
  - Au lieu de cela, les messages sont diffusés à travers un canal commun (souvent appelé sujet ou topic), et tous les abonnés intéressés par ces messages peuvent les recevoir.
1. **P1** : Il représente le producteur (ou l'éditeur) qui publie le message. Ce producteur envoie un message ( $m_1$ ) au canal.
  2. **Canal CH1** : C'est le canal (ou le sujet) auquel les messages sont publiés. Tous les messages envoyés à ce canal sont disponibles pour être consommés par les abonnés qui écoutent ce canal spécifique.
  3. **C1, C2, C3** : Ce sont les consommateurs (ou abonnés) qui se sont abonnés au canal CH1. Ils reçoivent le message  $m_1$  publié par P1.

## 2 - 9 - 1- Le pattern publication/abonnement (pub/sub).

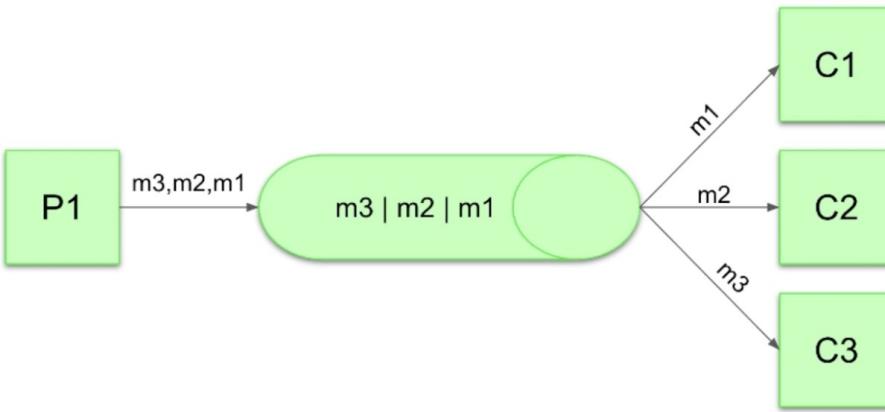
### Le modèle Publication / Abonnement

Les courtiers de messages qui prennent en charge ce modèle:

- **RabbitMQ**: Publish/Subscribe
- **ActiveMQ**: Broadcast
- **Kafka**: Topics (topics are always multi-subscriber)
- **NATS**: Publish-subscribe
- **AWS SQS/SNS**: Pub/sub via SNS
- **Azure Message Bus**: Topics and Subscriptions
- **Google Pub/Sub**: Publisher-Subscriber

## 2 - 9 - 1- Le pattern file d'attente de message

### Le modèle file d'attente de message



1. **P1** : C'est le producteur de messages qui envoie les messages dans la file d'attente.
2. **File d'attente de messages** : Représentée par une ellipse, c'est ici que les messages sont stockés jusqu'à ce qu'ils soient consommés. Les messages vont être traités dans l'ordre FIFO (First In, First Out) : le premier message envoyé (**m1**) sera le premier à être traité, suivi de **m2**, puis de **m3**.
3. **C1, C2, C3** : Ce sont les consommateurs qui reçoivent les messages de la file d'attente. Chaque consommateur prend un message de la file d'attente lorsque c'est possible. Dans ce schéma, **C1** prend **m1**, **C2** prend **m2** et **C3** prend **m3**.

## 2 - 9 - 2- Les patterns de gestion des données

### Le contexte et le problème :

*Quelle architecture pour les bases de données, pour une application à base de microservices, en tenant compte des contraintes suivantes :*

1. **Les services doivent** être faiblement couplés afin de pouvoir être développés, déployés et mis à l'échelle indépendamment.
2. **Certaines transactions d'affaires** doivent appliquer des invariants couvrant plusieurs services. Cela signifie qu'il faut maintenir la cohérence des données sur différents services qui peuvent être impliqués dans une transaction.
3. **D'autres transactions d'affaires** nécessitent d'interroger des données appartenant à plusieurs services, ce qui peut nécessiter des mécanismes de coordination ou de transaction distribuée.

## 2 - 9 - 2- Les patterns de gestion des données

### Le contexte et le problème :

*Quelle architecture pour les bases de données, pour une application à base de microservices, en tenant compte des contraintes suivantes :*

4. Certaines requêtes doivent joindre des données appartenant à plusieurs services, ce qui implique des défis en termes de performance et de cohérence des données.
5. Les bases de données doivent parfois être répliquées et partitionnées pour optimiser certaines requêtes, ce qui permet de répartir la charge et d'améliorer la disponibilité et la résilience du système.
6. Certains services peuvent nécessiter des exigences de stockage de données spécifiques; par exemple, l'utilisation de bases de données NoSQL comme MongoDB ou Neo4J peut être indiquée pour certains types de données ou de requêtes.

## 2 - 9 - 2- Les patterns de gestion des données

### 2 approches de solution

Une base de données par service

Une base de données partagées

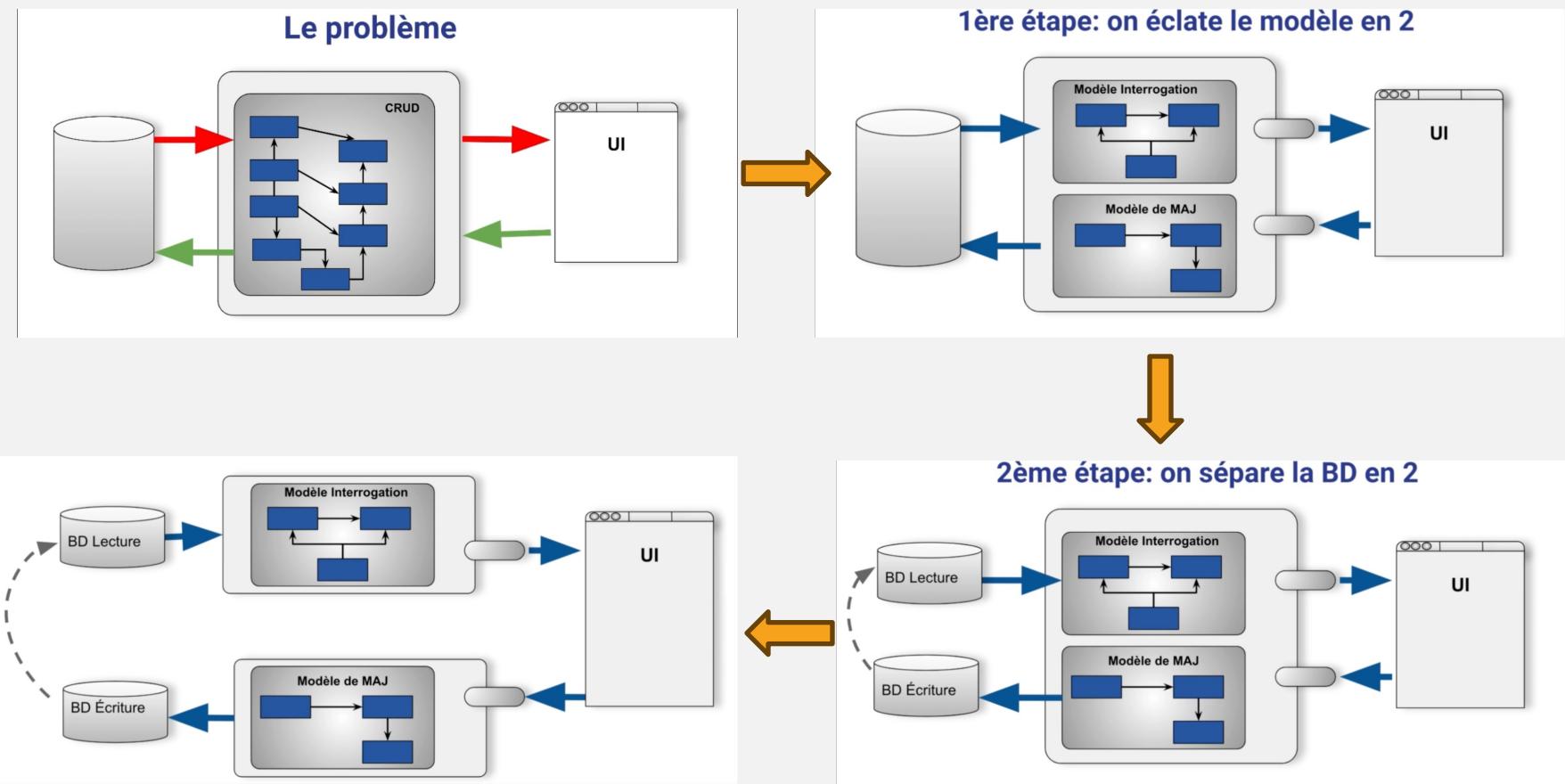
## 2 - 9 - 2- Le CQRS

- **CQRS signifie Command Query Responsibility Segregation : C'est un modèle d'architecture logicielle où la partie du système qui gère les commandes (écriture des données) est séparée de la partie qui gère les requêtes (lecture des données)**
- 1. **Séparation des modèles** : Dans CQRS, vous séparez le modèle de domaine en deux parties distinctes, l'une pour les commandes (écriture) et l'autre pour les requêtes (lecture). Cela permet de simplifier le modèle de conception, car chaque partie peut être optimisée pour ses responsabilités spécifiques.
- 2. **Écriture et lecture optimisées** : Les commandes peuvent impliquer des validations, des règles métiers, et des modifications d'état qui sont plus complexes et coûteuses en ressources. Les requêtes, d'autre part, peuvent être optimisées pour la vitesse et l'efficacité, souvent en utilisant des modèles de données dénormalisés qui reflètent directement les vues utilisateurs.
- 3. **Scalabilité et performance** : En séparant les responsabilités, vous pouvez scaler la lecture et l'écriture indépendamment, ce qui permet d'optimiser les performances et la disponibilité du système en fonction des besoins réels.

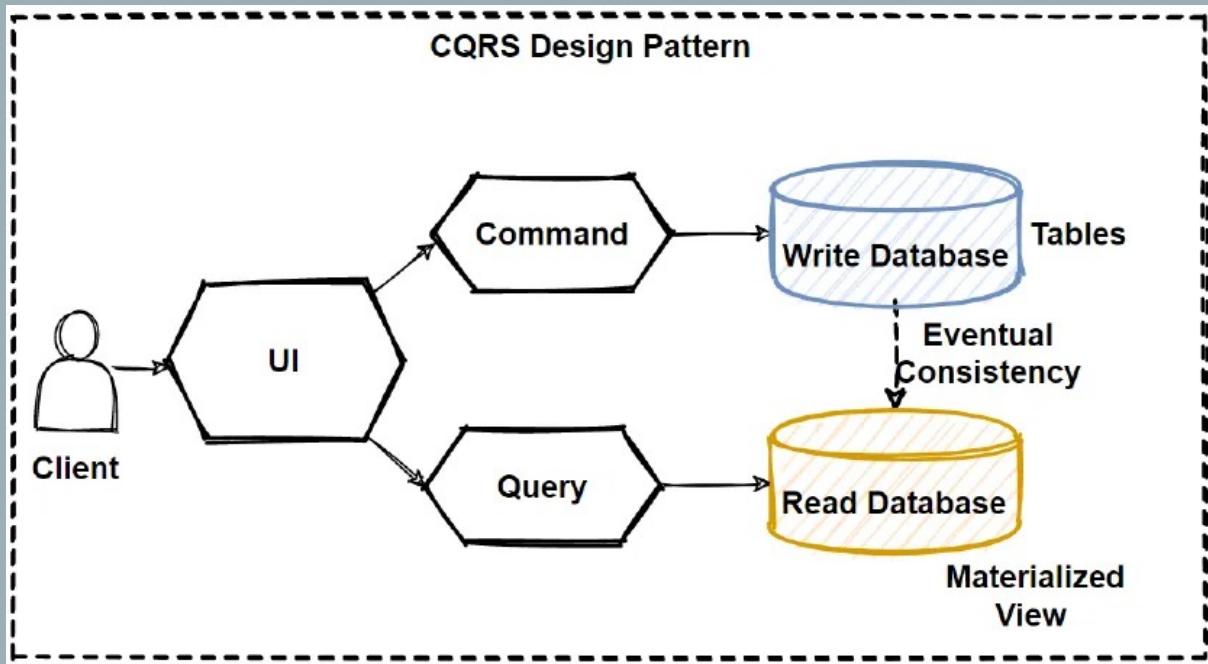
## 2 - 9 - 2- Le CQRS

4. **Simplification du code** : Cela peut simplifier le développement et la maintenance en séparant clairement la logique d'affaires complexe des opérations de lecture qui sont généralement plus simples.
5. **Consistance éventuelle** : CQRS est souvent associé à des modèles de consistance éventuelle, où les lectures peuvent ne pas refléter immédiatement les écritures. Il est nécessaire de gérer la synchronisation entre le côté commandes et le côté requêtes, ce qui peut introduire une complexité supplémentaire.
6. **Event Sourcing** : CQRS se marie bien avec le concept d'Event Sourcing, où les changements d'état sont stockés comme une séquence d'événements. Cela permet de reconstruire l'état actuel par la relecture des événements, et de créer facilement des états de projection optimisés pour les requêtes.

## 2 - 9 - 2- Le CQRS



## 2 - 9 - 1- Le pattern file d'attente de message



1. **P1** : C'est le producteur de messages qui envoie les messages dans la file d'attente.
2. **File d'attente de messages** : Représentée par une ellipse, c'est ici que les messages sont stockés jusqu'à ce qu'ils soient consommés. Les messages semblent être traités dans l'ordre FIFO (First In, First Out) : le premier message envoyé (m1) sera le premier à être traité, suivi de m2, puis de m3.
3. **C1, C2, C3** : Ce sont les consommateurs qui reçoivent les messages de la file d'attente. Chaque consommateur prend un message de la file d'attente lorsque c'est possible. Dans ce schéma, C1 prend m1, C2 prend m2 et C3 prend m3.

## 2 - 9 - 2- L'Event Sourcing

- **Event Sourcing** est un pattern architectural qui concerne la persistance de l'état d'une application.
- Au lieu de simplement stocker l'état actuel des entités de données dans une base de données, l'Event Sourcing implique de stocker une séquence d'événements qui reflètent toutes les modifications apportées à l'état de l'application au fil du temps.

## 2 - 9 - 2- L'Event Sourcing

Voici comment fonctionne l'Event Sourcing :

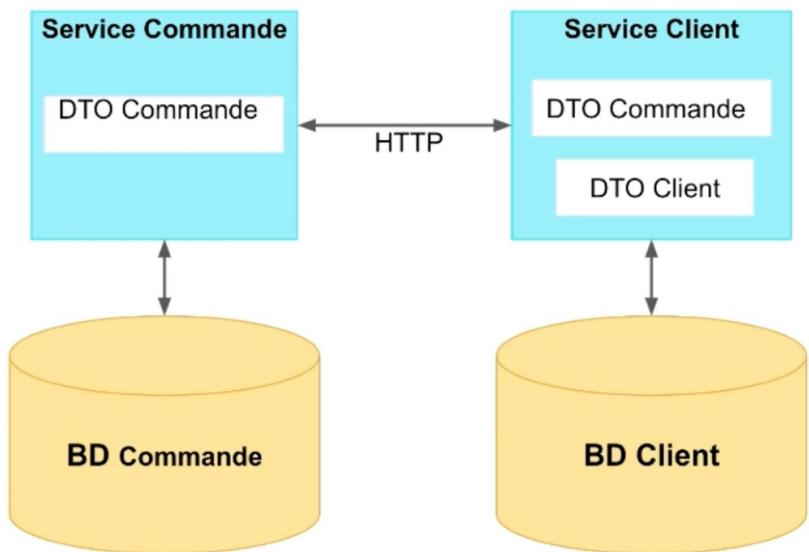
1. **Événements** : Au cœur du système se trouvent les événements, qui sont des objets immuables enregistrant le fait qu'une action s'est produite, ainsi que tous les détails pertinents de cette action.
2. **Stockage d'Événements** : Ces événements sont stockés dans un journal d'événements dans l'ordre chronologique dans lequel ils se produisent.
3. **Reconstruction d'État** : L'état actuel de l'application peut être reconstruit à tout moment en rejouant les événements depuis le début.
4. **Projections** : Pour les requêtes, des "projections" peuvent être construites à partir du journal des événements, qui sont des vues de l'état optimisées pour certaines opérations de lecture.

## 2 - 9 - 2- L'Event Sourcing

Voici comment fonctionne l'Event Sourcing :

1. **Événements** : Au cœur du système se trouvent les événements, qui sont des objets immuables enregistrant le fait qu'une action s'est produite, ainsi que tous les détails pertinents de cette action.
2. **Stockage d'Événements** : Ces événements sont stockés dans un journal d'événements dans l'ordre chronologique dans lequel ils se produisent.
3. **Reconstruction d'État** : L'état actuel de l'application peut être reconstruit à tout moment en rejouant les événements depuis le début.
4. **Projections** : Pour les requêtes, des "projections" peuvent être construites à partir du journal des événements, qui sont des vues de l'état optimisées pour certaines opérations de lecture.

## 2 - 9 - 2- L'Event Sourcing



### Limites d'une telle architecture :

- **Couplage** : Même si les services sont conçus pour être indépendants, ils peuvent développer un couplage temporel ou fonctionnel. Par exemple, si le Service Client doit attendre une réponse du Service Commande pour compléter une tâche, il y a un couplage temporel.
- **Complexité de gestion des transactions** : Les transactions qui nécessitent des opérations coordonnées entre les services peuvent être difficiles à gérer, en particulier pour maintenir la cohérence des données entre les services.
- **Consistance des données** : Chaque service ayant sa propre base de données, cela peut entraîner des problèmes de cohérence des données, surtout si les mêmes données sont dupliquées dans plusieurs services.

## 2 - 9 - 2- L'Event Sourcing

### Mise en place de l'event sourcing :

#### 1. Service Commande :

Ce microservice gère les commandes. Il communique avec une base de données dédiée, ici nommée "BD Commande", où il stocke et récupère des informations liées aux commandes des clients.

#### 2. Service Client :

Ce microservice gère les informations relatives aux clients. Il interagit avec sa propre base de données, "BD Client", pour gérer les informations client telles que les profils, les préférences, et l'historique des interactions.

#### 3. Messagerie :

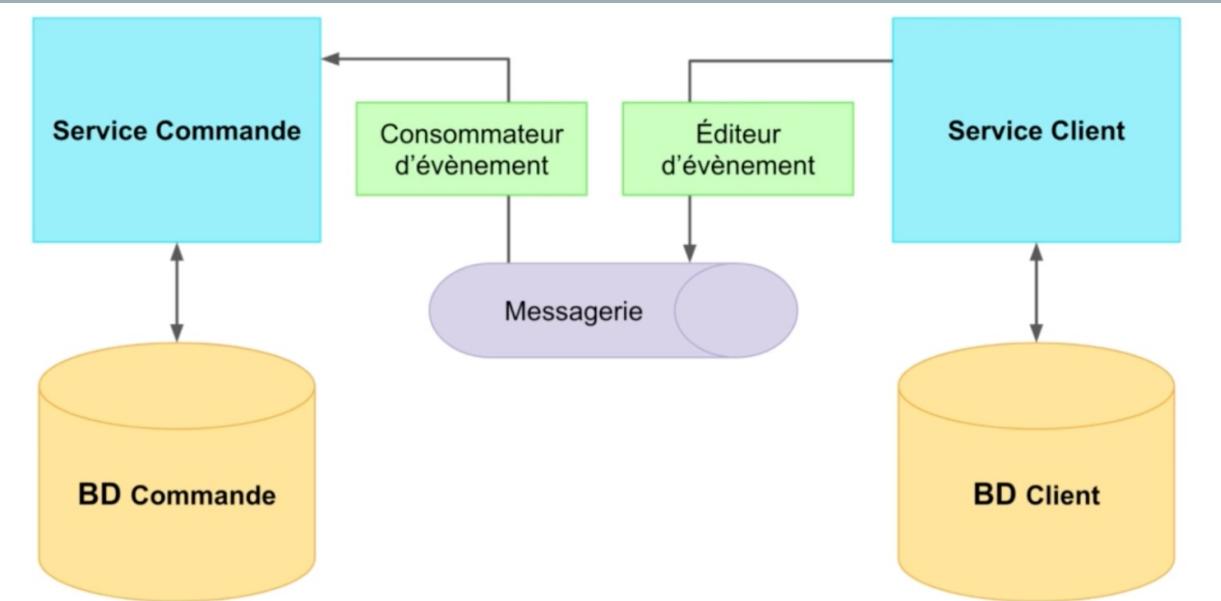
C'est un système de messagerie ou un bus d'événements. Son rôle est de permettre une communication asynchrone entre les microservices. Cela signifie que les services peuvent émettre ou consommer des messages sans dépendre directement les uns des autres ou sans attendre une réponse immédiate.

#### 4. Consommateur d'événement :

Il s'agit d'un composant ou d'un service qui écoute les événements publiés sur la messagerie. Quand un événement d'intérêt est détecté, ce consommateur réagit en conséquence, par exemple, en mettant à jour des données dans la BD Commande ou en déclenchant une autre action au sein du système.

#### 5. Éditeur d'événement :

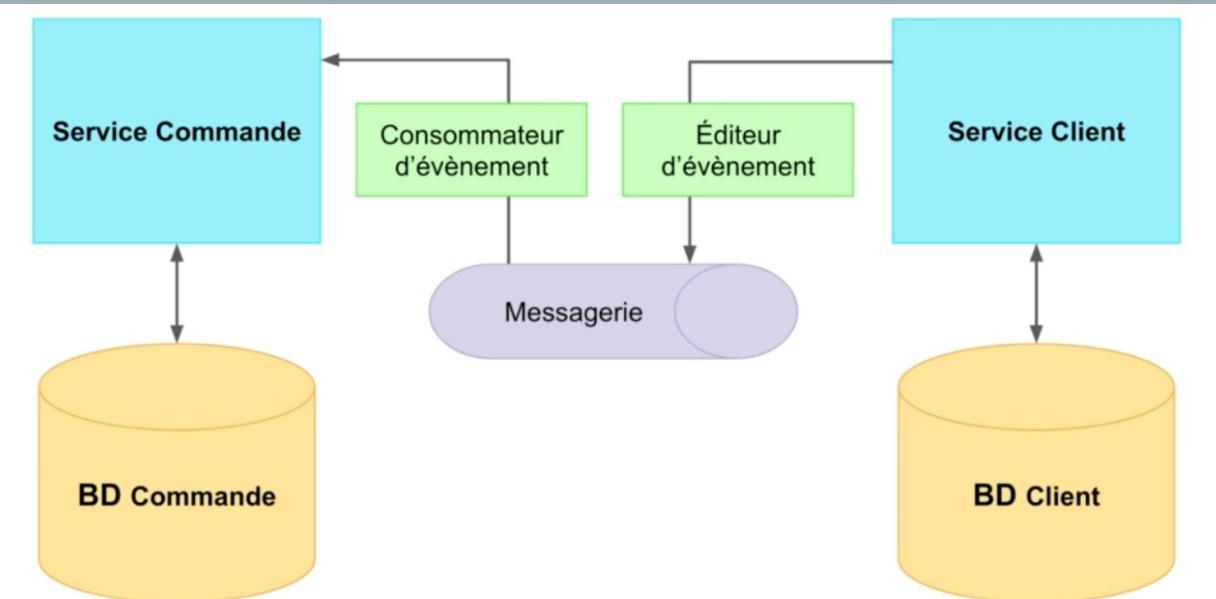
C'est le composant qui publie les événements dans la messagerie. Par exemple, lorsque le Service Commande effectue une opération qui nécessite de notifier d'autres parties du système (comme le Service Client), il publie un événement sur la messagerie.



## 2 - 9 - 2- L'Event Sourcing

### Mise en place de l'event sourcing :

- Quand une commande est passée via le Service Commande, ce service effectue les opérations nécessaires dans la BD Commande.
- Une fois la commande traitée, le Service Commande publie un événement dans la messagerie via l'Éditeur d'événement, indiquant qu'une nouvelle commande a été passée ou mise à jour.
- Le Consommateur d'événement écoute ces messages. Lorsqu'il détecte un événement pertinent, il effectue les actions nécessaires, comme mettre à jour le statut du client dans la BD Client.
- Le Service Client, en cas de besoin, peut également être informé via ces événements et peut réagir en conséquence, par exemple, en mettant à jour les informations de profil d'un client basées sur son activité de commande.

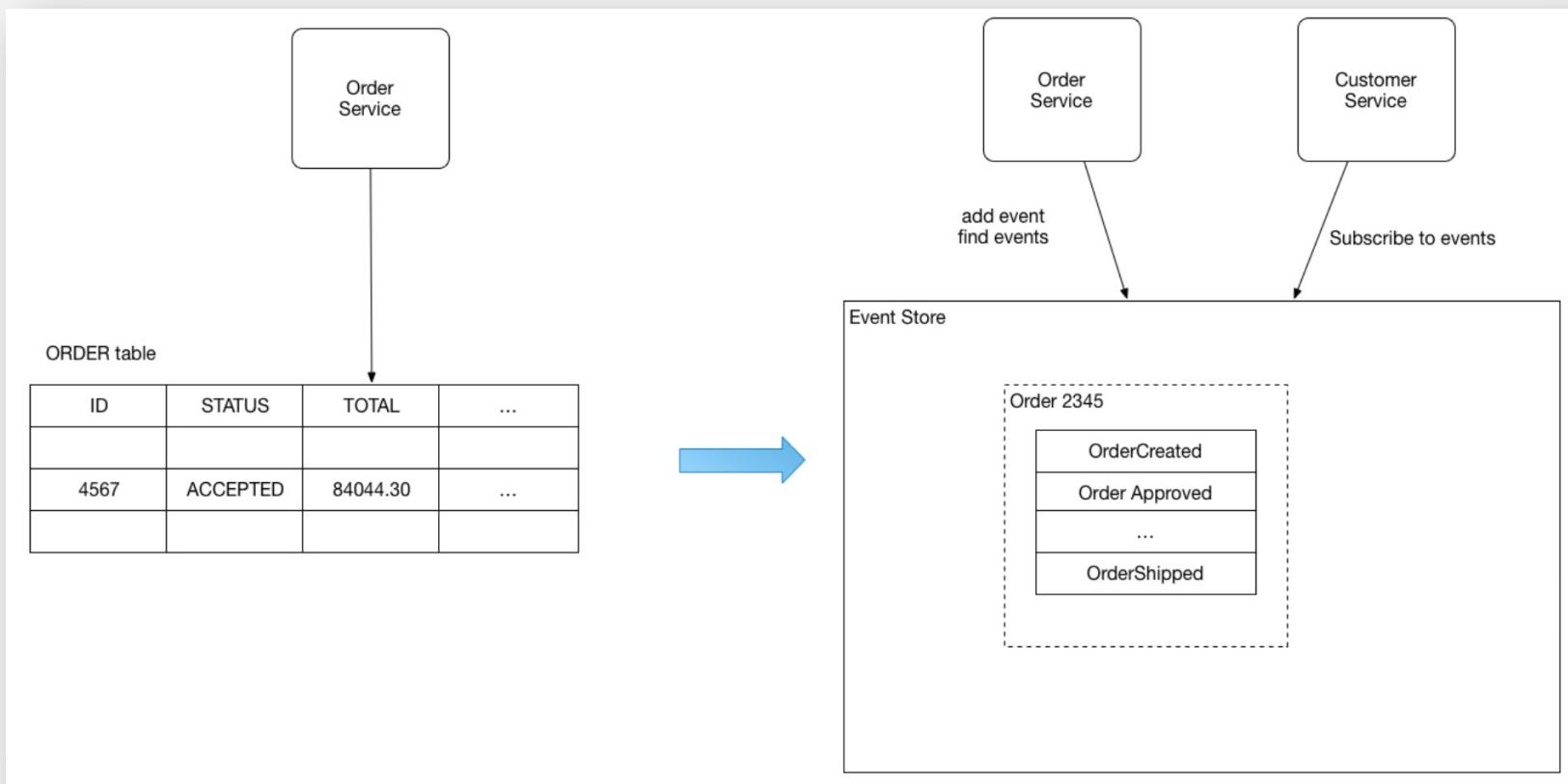


## 2 - 9 - 2- L'Event Sourcing

Quel outil pour mettre en place cet event sourcing :

1. **Apache Kafka:** C'est une plateforme de streaming distribuée qui peut gérer efficacement les flux d'événements. Kafka est souvent utilisé pour construire des architectures robustes d'Event Sourcing grâce à sa capacité à traiter de grandes quantités de messages en temps réel.
2. **EventStore:** C'est une base de données spécialisée pour le stockage d'événements. Elle est conçue spécifiquement pour le scénario d'Event Sourcing et offre des fonctionnalités comme les projections pour traiter les événements de manière asynchrone.
3. **RabbitMQ/ActiveMQ:** Ce sont des courtiers de messages qui peuvent être utilisés pour mettre en œuvre la messagerie asynchrone requise pour l'Event Sourcing, bien qu'ils soient plus traditionnellement utilisés pour la messagerie que pour le stockage des événements lui-même.
4. **Redis Streams:** Redis offre une structure de données de type stream qui peut être utilisée pour une mise en œuvre simple de l'Event Sourcing, surtout lorsque les exigences en termes de persistance sont moins strictes.
5. **Amazon Kinesis:** Un service de streaming de données en temps réel géré par AWS, qui peut être utilisé pour collecter, stocker et analyser des flux d'événements à grande échelle.
6. **Azure Event Hubs:** Une plateforme de streaming de données à grande échelle de Microsoft Azure qui peut être utilisée pour construire des solutions d'Event Sourcing.
7. **Google Pub/Sub:** Un service de messagerie asynchrone qui permet de publier et de consommer des messages entre des applications indépendantes, utilisable pour l'Event Sourcing sur Google Cloud.

## 2 - 9 - 2- L'Event Sourcing



## 2 - 9 - 2- L'Event Sourcing

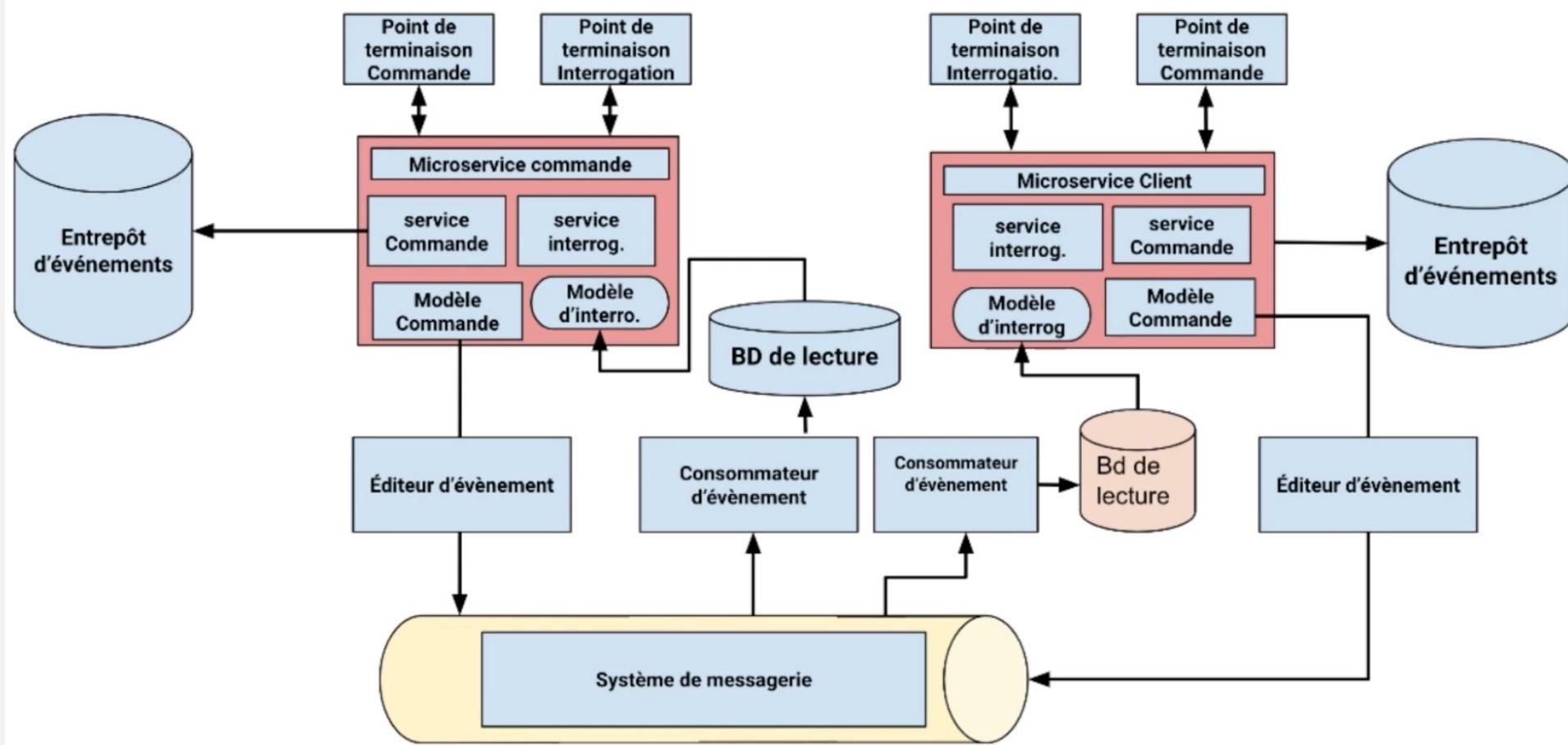
### Les limites :

1. C'est un style de programmation différent qui nécessite une courbe d'apprentissage.
2. La capacité de l'entrepôt d'événements doit être plus grande pour stocker toute l'historique des enregistrements.
3. L'entrepôt d'événements est difficile à interroger car il nécessite des requêtes qui impliquent des manipulations complexes pour reconstruire l'état des entités d'affaires.

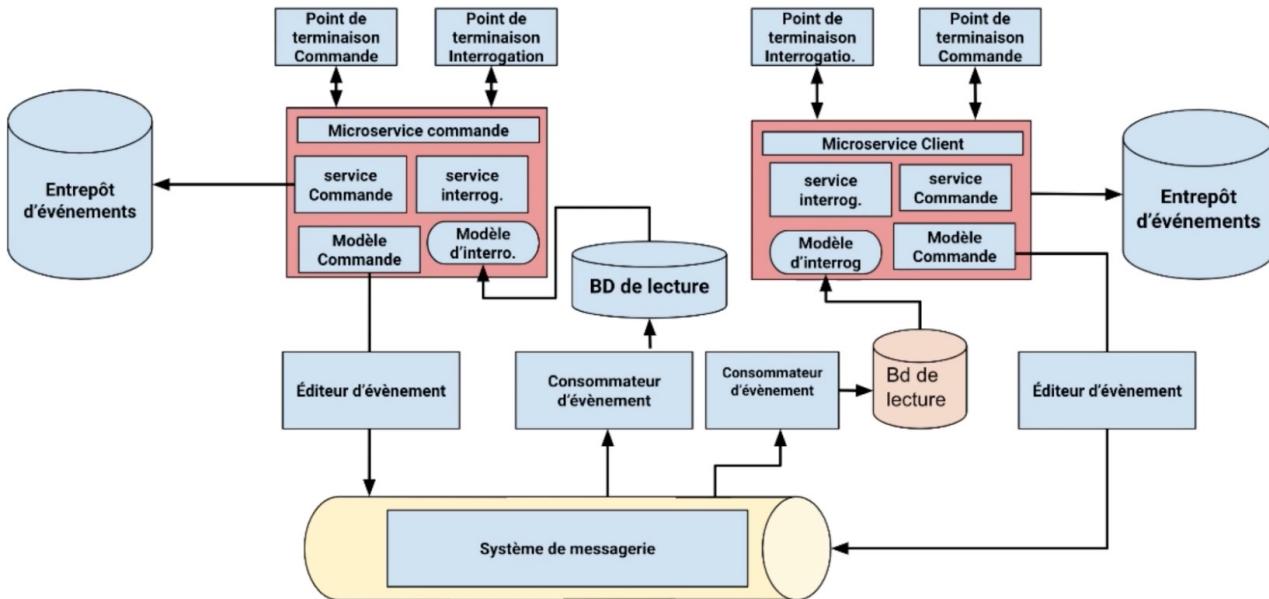
## 2 - 9 - 3 - La combinaison CQRS + Event Sourcing

- La combinaison de **CQRS** (Command Query Responsibility Segregation) et d'**Event Sourcing** est une architecture puissante souvent utilisée dans les systèmes distribués pour séparer les préoccupations de lecture et d'écriture des données et pour fournir un historique immuable des changements d'état dans le système.

## 2 - 9 - 2 - La combinaison CQRS + Event Sourcing



## 2 - 9 - 2- La combinaison CQRS + Event Sourcing



### Mise en place de l'event sourcing :

- Quand une commande est passée via le Service Commande, ce service effectue les opérations nécessaires dans la BD Commande.
- Une fois la commande traitée, le Service Commande publie un événement dans la messagerie via l'Éditeur d'événement, indiquant qu'une nouvelle commande a été passée ou mise à jour.
- Le Consommateur d'événement écoute ces messages. Lorsqu'il détecte un événement pertinent, il effectue les actions nécessaires, comme mettre à jour le statut du client dans la BD Client.
- Le Service Client, en cas de besoin, peut également être informé via ces événements et peut réagir en conséquence, par exemple, en mettant à jour les informations de profil d'un client basées sur son activité de commande.

## 2 - 9 - 2 - Le pattern SAGA

### Le contexte :

1. Les transactions doivent être Atomiques, Cohérentes, Isolées et Durables (ACID) **Transaction ACID** et Architecture multiservices:
  - L'**atomicité** est un ensemble indivisible et irréductible d'opérations qui doivent toutes être exécutées (ou toutes ne pas être exécutées)
  - La **cohérence** signifie que la transaction ne doit faire passer les données que d'un état valide à un autre état valide
  - L'**isolement** garantit que si les transactions sont exécutées simultanément, elles produisent le même état de données que si elles étaient exécutées séquentiellement
  - La **durabilité** garantit que les transactions qui sont validées demeurent valides même en cas de panne du système ou de panne de courant
2. Les transactions au sein d'un même service sont **ACID**

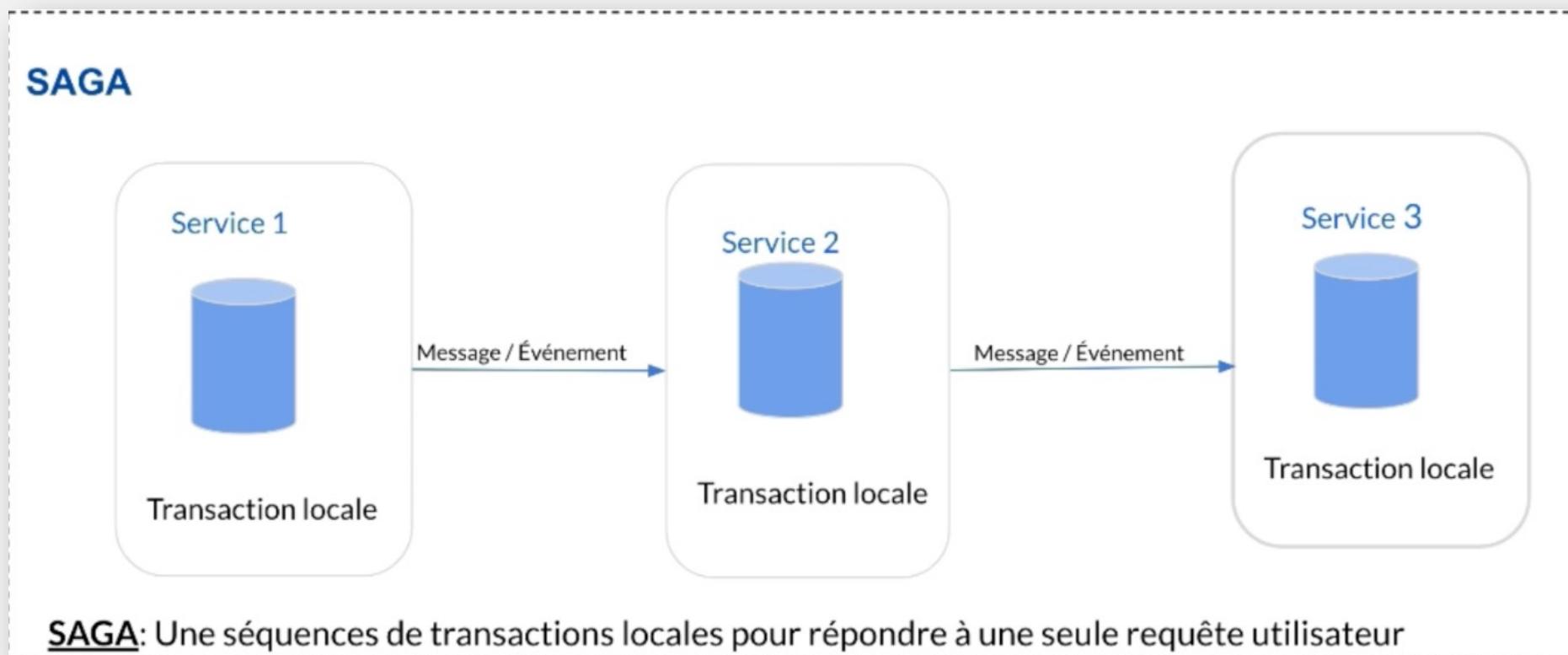
## 2 - 9 - 2 - Le pattern SAGA

### Le problème :

- Un modèle "**une Base de données par microservice**" offre de nombreux avantages
- Cependant, garantir la cohérence des données dans les bases de données qui sont spécifiques aux services pose des problèmes
- On peut toujours envisager d'utiliser les mécanismes de gestion des transactions distribuées comme le protocole de validation en deux phases
- Une autre limitation des transactions distribuées est la synchronisation des communications interprocessus et la disponibilité

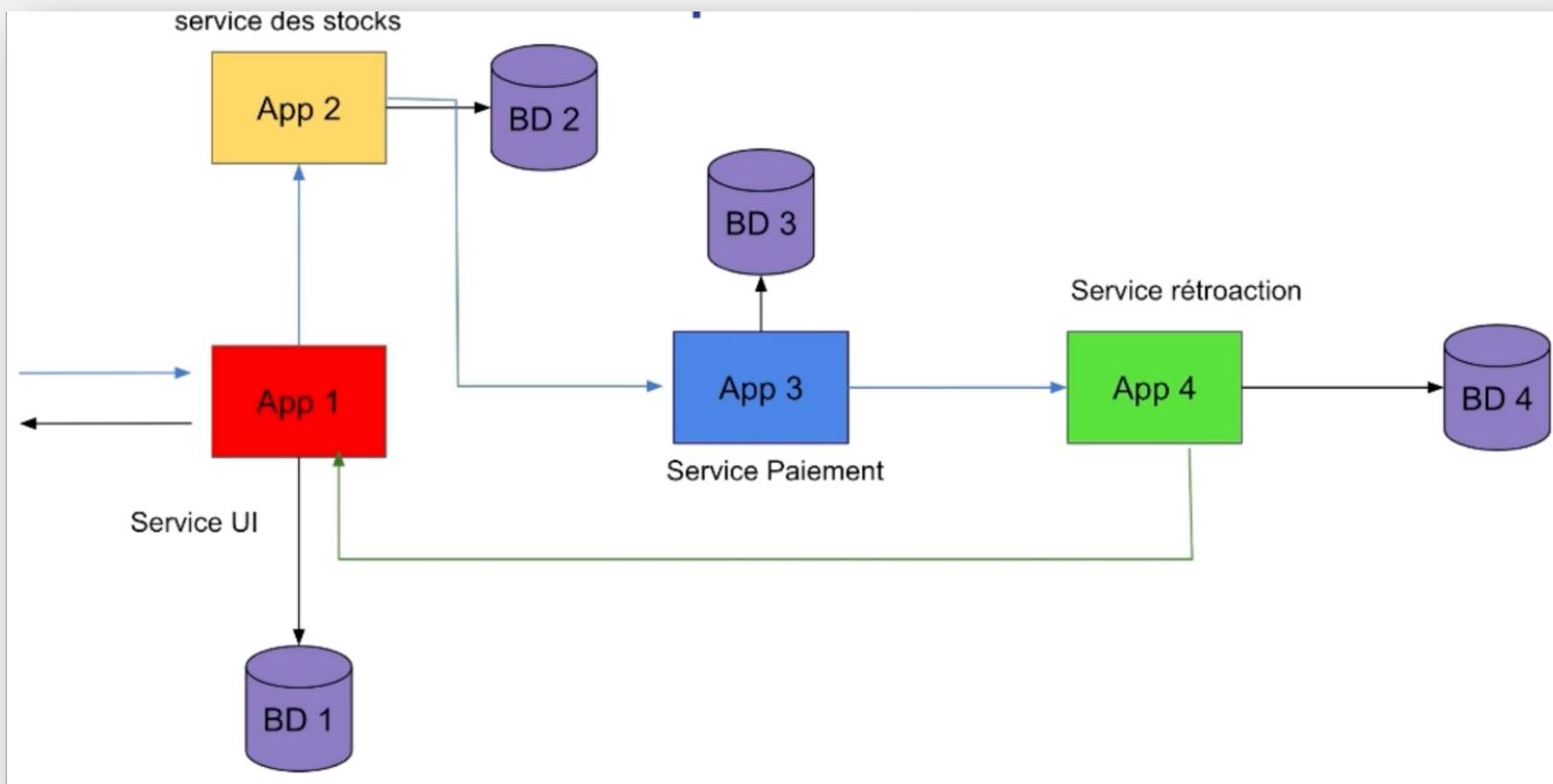
## 2 - 9 - 2 - Le pattern SAGA

### Le solution :



## 2 - 9 - 2 - Le pattern SAGA

Exemple :

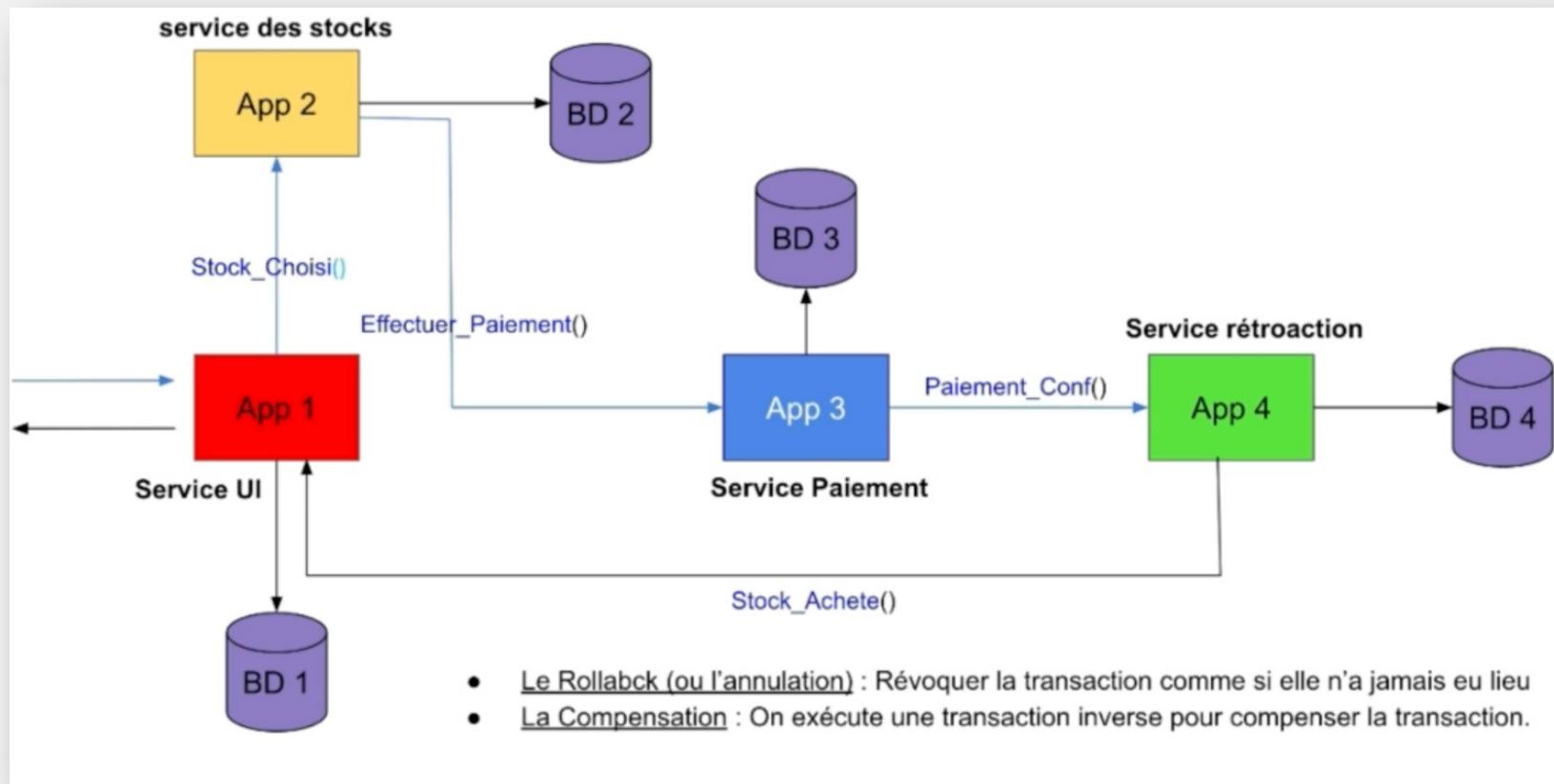


2 approches  
d'implémentations :

- **Approche basée sur les évènements.**
- **Approche basée sur les commandes.**

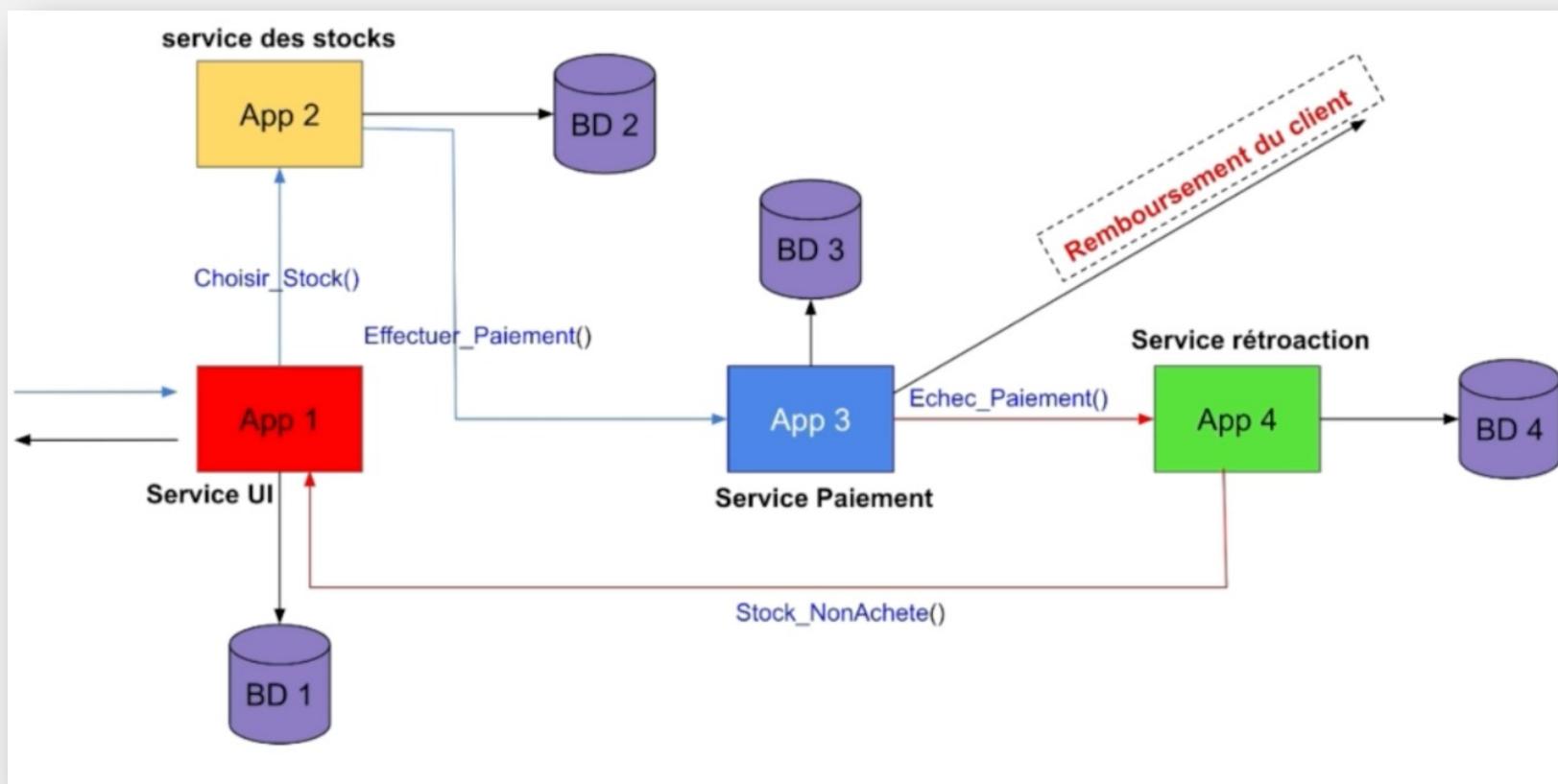
## 2 - 9 - 2 - Le pattern SAGA

### Exemple : Approche chorégraphie



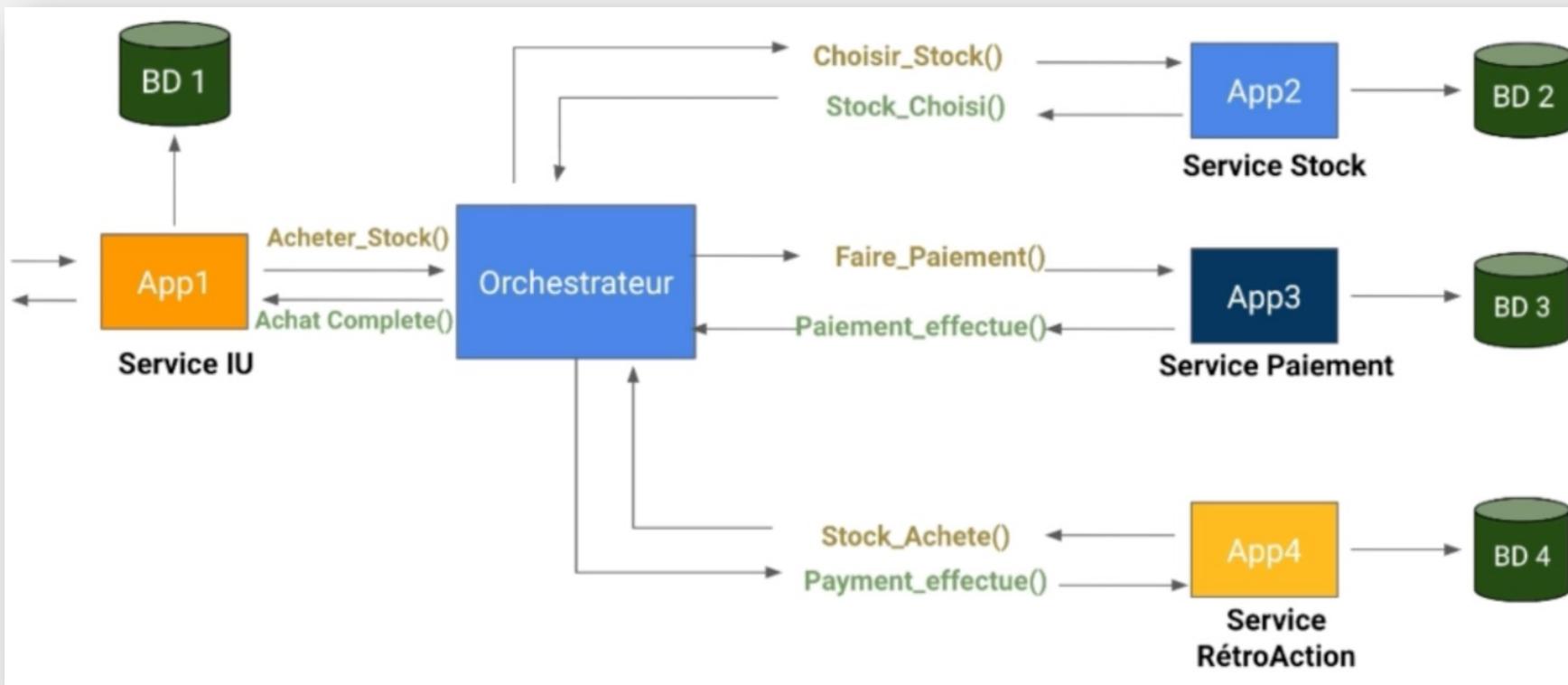
## 2 - 9 - 2 - Le pattern SAGA

### Exemple : Approche chorégraphie (en cas d'échec)



## 2 - 9 - 2 - Le pattern SAGA

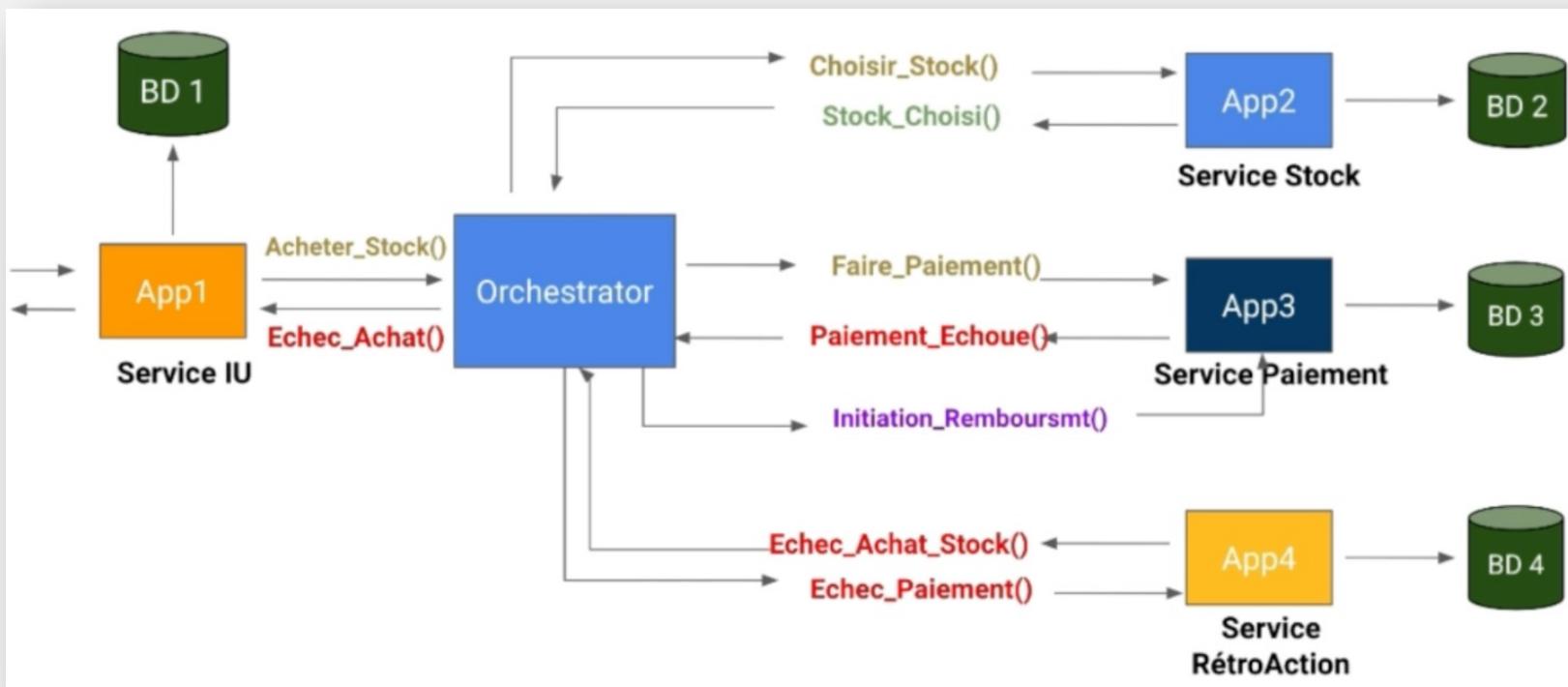
### Exemple : Approche par orchestrateur



Un exemple d'outil qui peut être utilisé comme orchestrateur dans ce contexte est **Camunda BPM** avec son moteur de règles et son système de gestion de workflow, peut orchestrer des sagas en coordonnant les différentes étapes et compensations nécessaires.

## 2 - 9 - 2 - Le pattern SAGA

### Exemple : Approche par orchestrateur (echec et compensation)



Un exemple d'outil qui peut être utilisé comme orchestrateur dans ce contexte est **Camunda BPM** avec son moteur de règles et son système de gestion de workflow, peut orchestrer des sagas en coordonnant les différentes étapes et compensations nécessaires.

## 2 - 9 - 2 - Le pattern SAGA

### Les avantages

- Utiliser pour maintenir la consistance dans le cadre des transactions distribuées dans une application à base de microservices.
- Dans l'application de ce pattern, les microservices ne sont pas bloqués les uns par les autres.

### Les désavantages

- L'orchestrateur ajoute une surcharge de travail
- Difficile à débugger les transactions qui sont distribuées par nature
- Implémentation des actions de compensation en cas d'échec des transactions
- Il faut aussi penser à la situation où l'action de compensation échoue

## 2 - 9 - 2 - Les patterns de gestion des données

### Résumé et Rappel :

- Le **pattern CQRS** permet de séparer les responsabilités entre les opérations d'écriture et les opérations d'interrogation des données dans deux microservices afin d'optimiser spécifiquement chacun d'eux
- Le **pattern Event-Sourcing** permet d'enregistrer les événements qui changent l'état de l'application dans un entrepôt. Ces événements peuvent être rejoués pour retrouver l'état courant d'une entité. Le modèle inclut un courtier de messages auquel d'autres services peuvent s'abonner.
- Une **combinaison des patterns CQRS et Event-Sourcing** permet de résoudre certains cas d'utilisation. En particulier, on peut utiliser le modèle Event-sourcing pour implémenter les mécanismes de mise à jour des vues matérialisées dans la base de données de lecture d'un modèle CQRS, à partir des données de l'entrepôt d'écriture.
- Le **pattern SAGA** fournit une gestion des transactions à l'aide d'une séquence de transactions locales.

## 2 - 9 - 3 - Le pattern de fiabilité

### Problème et contexte :

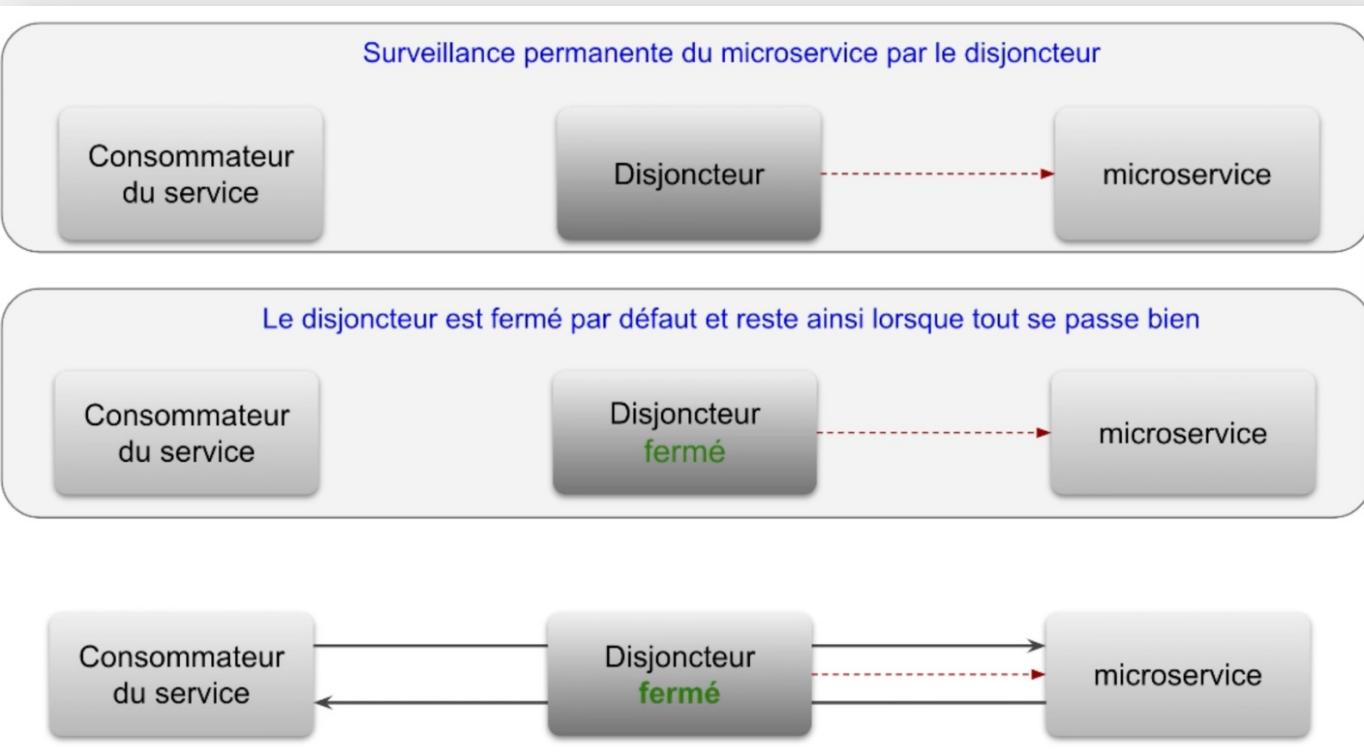
Lorsqu'un client communique avec un microservice distant, on a en permanence deux situations à gérer:

1. **La disponibilité du service** : Est-ce qu'on peut accéder au service.
2. **La réactivité du service** : Une fois qu'on a accès au service, est-ce qu'on a une réponse dans un délai acceptable

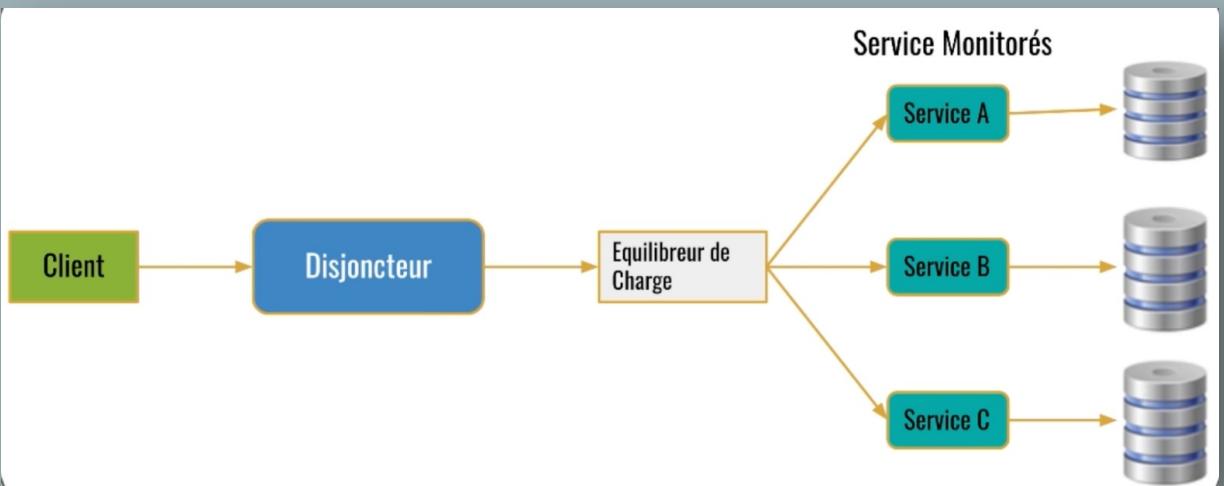


**Le pattern disjoncteur**

## 2 - 9 - 2- Le pattern de fiabilité



## 2 - 9 - 3- Le pattern de fiabilité



Voici les types les plus courants de patterns de disjoncteurs :

- 1. Disjoncteur à état simple :**
  - **Fermé** : L'état normal du circuit où les appels passent normalement.
  - **Ouvert** : Lorsque les échecs dépassent un certain seuil, le disjoncteur s'ouvre pour empêcher les appels et permet au système sous-jacent de se rétablir.
  - **Demi-ouvert** : Après un temps prédéfini en état ouvert, le disjoncteur passe en état demi-ouvert où un nombre limité d'appels est tenté pour tester si le problème sous-jacent a été résolu.
- 2. Disjoncteur à état temporisé :**
  - Dans ce cas, le disjoncteur reste ouvert seulement pour un temps déterminé avant de tenter automatiquement de se réinitialiser.
- 3. Disjoncteur à compteur d'erreurs :**
  - Cela utilise un compteur d'erreurs pour suivre le nombre de défaillances. Une fois un seuil d'erreur atteint, le disjoncteur s'ouvre.
- 4. Disjoncteur à jauge d'erreur :**
  - Plutôt que de compter simplement les échecs, une jauge ou un pourcentage des appels échoués sur le total des appels est utilisé pour décider quand ouvrir le disjoncteur.
- 5. Disjoncteur adaptatif :**
  - Ces disjoncteurs peuvent ajuster leurs seuils en temps réel en fonction de la performance passée du système ou d'autres métriques opérationnelles.
- 6. Disjoncteur manuel :**
  - Un opérateur humain peut manuellement ouvrir ou fermer le disjoncteur, souvent utilisé pour des interventions planifiées ou en réponse à des alertes surveillées.