Architectures Avancées

Chapitre 1:Architectures Événementielles

- Introduction aux architectures réactives
- Event-Driven Architecture avec Apache Kafka
- CQRS et Event Sourcing

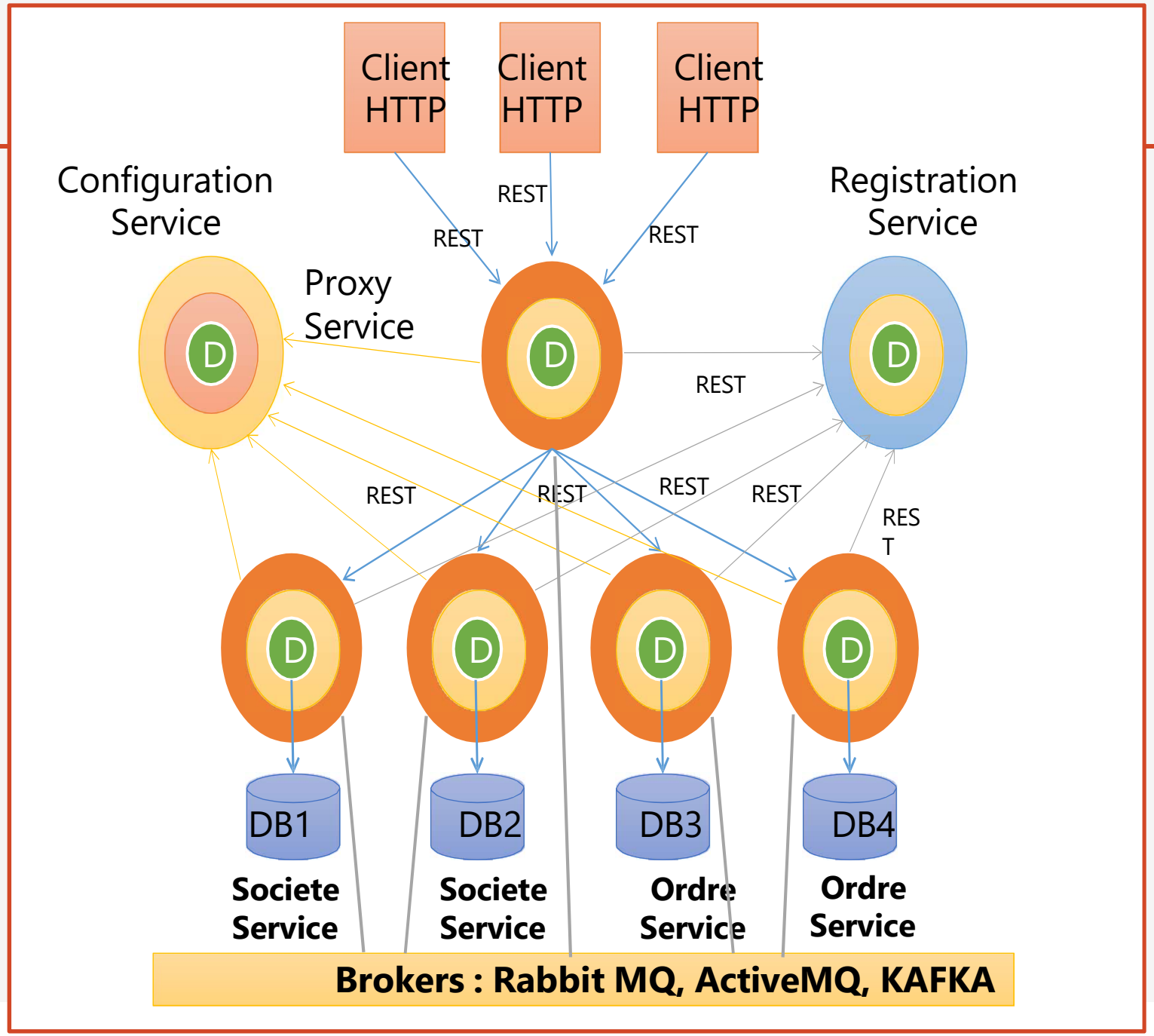
Introduction

Les applications modernes créées à l'aide de **microservices** utilisent souvent une **architecture orientée événements**, qui s'appuie sur des événements pour initier et faciliter la communication entre des services **découplés**.

L'ajout d'un article dans un panier d'achat sur un site de commerce électronique est un exemple d'événement ; il s'agit d'un **changement d'état** ou d'une **mise à jour**. Les événements peuvent soit être de simples **identifiants**, soit **transporter l'état**, comme l'article acheté, son prix et l'adresse de livraison, ou encore une **notification** indiquant qu'une commande a été expédiée.

Les architectures orientées événements sont constituées de **trois éléments principaux** : les **producteurs d'événements**, les **routeurs d'événements** et les **consommateurs d'événements**. Un événement est publié par un producteur et envoyé au routeur, qui le filtre et le transmet aux consommateurs.

Grâce au **découplage** entre les services producteurs et consommateurs, chacun d'eux peut être **développé, modifié et déployé indépendamment**.


Client HTTP
Client HTTP
Client HTTP
Configuration Service
Proxy Service
Registration Service
REST
REST
REST
REST
REST
REST
REST
REST
RES T
D
DB1
DB2
DB3
DB4
Societe Service
Societe Service
Ordre Service
Ordre Service
Brokers : Rabbit MQ, ActiveMQ, KAFKA

Architecture Orientée Événements (EDA)

Dans les applications à microservices

Exemple d'Événement

Article ajouté au panier

Item ajouté | 25 € | Adresse de livraison

Producteur d'Événements

Émet un événement

Événement

Publie l'événement

Routeur d'Événements

Filtre & Distribue

Consommateurs d'Événements

Service Paiement

Service Livraison

Service Notification

Rèagissent à l'événement

Services Découplés: Évoluent & Se Déploient Séparément

Definitions

Une **architecture orientée événements** est un style architectural où les **composants du système communiquent principalement via des événements**, plutôt que par des appels directs synchrones (comme dans une architecture classique de type REST ou SOA).

Définition

•Un **événement** est un changement d'état ou une action qui a eu lieu dans le système, par exemple :

- CommandeCréée
- PaiementEffectué
- UtilisateurInscrit

•Les **producteurs d'événements** émettent ces événements.

•Les **consommateurs d'événements** les écoutent et agissent en conséquence.

Qu’est-ce qu’un événement ?

- Un événement est **tout fait marquant ou changement d’état** concernant un logiciel ou un matériel système.
- Un message ou une notification envoyé par le système pour informer un autre composant qu’un événement s’est produit **n’est pas le même chose qu’un événement**.
- Un événement peut avoir pour origine des **entrées internes ou externes** :

Utilisateur : par exemple une frappe au clavier ou un clic de souris

Source externe : par exemple la sortie d’un capteur

Système : par exemple le chargement d’un programme

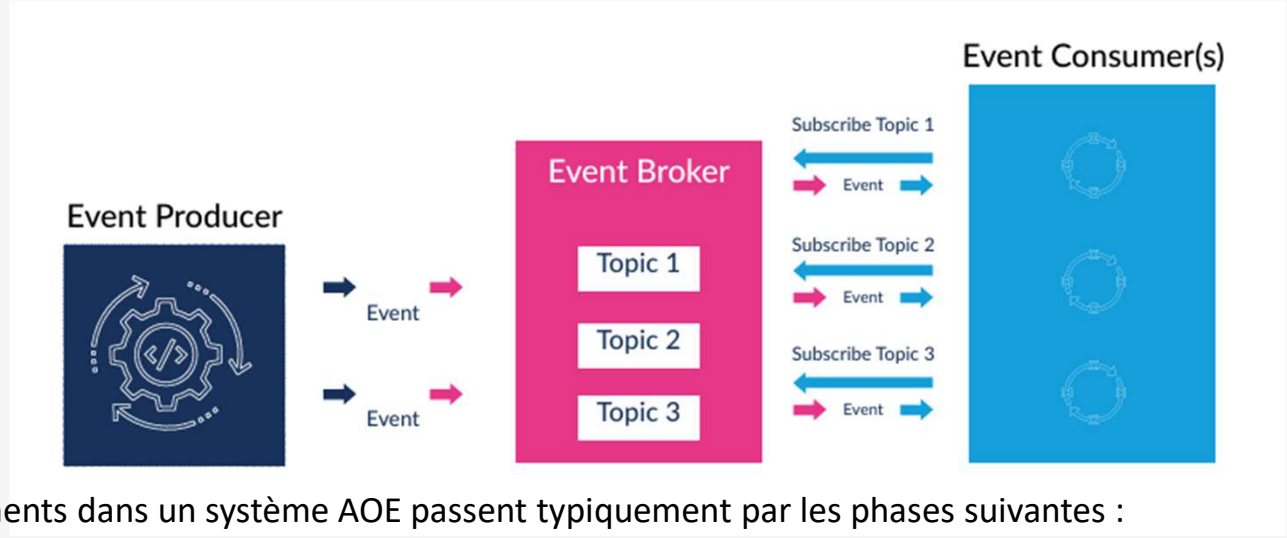
Composants clés d'une architecture orientée événements

1. **Event Producer (Producteur d'événements)**
    1. Génère des événements lorsqu'une action ou un changement se produit.
    2. Exemple : un service de commandes qui émet un événement CommandeCréée.
2. **Event Broker / Event Bus (Courtier d'événements)**
    1. Intermédiaire qui transporte les événements du producteur vers les consommateurs.
    2. Exemples : **Kafka**, **RabbitMQ**, **ActiveMQ**.
3. **Event Consumer (Consommateur d'événements)**
    1. Écoute les événements et déclenche des actions.
    2. Exemple : un service de facturation qui réagit à CommandeCréée pour générer une facture.
4. **Event Store (optionnel)**
    1. Stocke l'historique des événements pour l'audit ou la reconstruction d'état.
    2. Exemples : **Event Sourcing**.

Qu’est-ce que l’architecture événementielle ?

Une organisation peut identifier des **« événements »** ou des **moments clés du métier** (tels qu’une transaction, une visite du site, l’abandon d’un panier d’achat, etc.) en utilisant une **architecture orientée événements (Event-Driven Architecture – EDA)**, qui est un **modèle de conception logicielle**, et y **réagir en temps réel ou quasi en temps réel**.

L’ancienne architecture **« requête/réponse »**, dans laquelle les services devaient attendre une réponse avant de passer à la tâche suivante, est remplacée par ce modèle. Les **événements** contrôlent le flux de l’architecture orientée événements, laquelle est conçue pour **réagir aux événements** ou **déclencher des actions** en réponse à un événement.



Les événements dans un système AOE passent typiquement par les phases suivantes :
Création : Un producteur génère un événement suite à un changement d'état.
Envoi : L'événement est envoyé à un broker.
Routage : Le broker délivre l'événement aux consommateurs abonnés.
Traitement : Les consommateurs réagissent à l'événement, entraînant potentiellement la création de nouveaux événements.

Caractéristiques principales

- **Asynchrone** : le producteur ne bloque pas le consommateur.
- **Découplage** : producteur et consommateur ne se connaissent pas directement.
- **Scalabilité** : facile à ajouter de nouveaux consommateurs ou producteurs.
- **Réactivité** : le système réagit en temps réel aux changements.

La communication asynchrone

La **communication asynchrone** est un terme couramment utilisé pour décrire l'**architecture orientée événements**. Ainsi, ni l'émetteur ni le destinataire n'ont besoin d'attendre que l'autre termine sa tâche en cours. Ce n'est pas un message particulier qui pilote directement les systèmes.

Par exemple, un **appel téléphonique** est considéré comme une communication **synchrone**, ou similaire au modèle traditionnel **« requête/réponse »**. Vous recevez un appel de quelqu'un qui vous demande d'exécuter une tâche. Vous attendez pendant que l'autre personne termine, puis les deux parties raccrochent.

À l'inverse, la **messagerie texte** est un exemple d'événement **asynchrone**. Lorsque vous envoyez un message, vous pouvez ne même pas savoir qui est le destinataire ni si quelqu'un est à l'écoute, et vous **n'attendez pas de réponse immédiate**.

Comment l'architecture orientée événements est développée

Ces dernières années, on a observé un **glissement** de l'accent mis par l'architecture orientée services (**SOA**) sur les **données au repos** vers une focalisation sur les **événements** (architecture orientée événements). En s'éloignant de l'**agrégation des données** et des **lacs de données**, l'attention se porte désormais sur les **données en transit** (*data in flight*) et sur leur suivi lorsqu'elles circulent d'un point à un autre.

La majorité des systèmes fonctionnent selon ce que l'on appelle le **modèle centré sur les données**, dans lequel les **données constituent l'autorité ultime**.

Microservices et architecture orientée événements

Définition des microservices
Les **microservices** sont une approche où une application est divisée en **petits services indépendants**, chacun ayant sa propre responsabilité métier.
Chaque microservice est autonome, peut être développé, déployé et scalé indépendamment.

Lien avec l'EDA
Les microservices ont besoin de **communiquer entre eux**.
L'EDA est idéale pour les microservices car elle permet :

- Un **découplage fort** : chaque microservice peut réagir aux événements sans connaître les autres services.
- Une **scalabilité indépendante** : un microservice peut être mis à l'échelle sans impacter les autres.
- Une **réactivité en temps réel** : les événements sont traités dès qu'ils surviennent.

Exemple concret
1. Service **Commande** émet **CommandeCréée**.
2. Service **Paiement** consomme cet événement et effectue le paiement.
3. Service **Notification** consomme le même événement pour envoyer un email au client.
4. Chaque microservice reste indépendant et communique uniquement via les événements

Avantages de l'EDA pour les microservices

• **Découplage fort** entre services

• **Réduction des points de blocage** (asynchrone)

La **scalabilité horizontale** (ou *scaling horizontal*) désigne la capacité d'un système à **supporter une charge croissante en ajoutant de nouvelles instances**, plutôt qu'en renforçant une seule machine.

•**Traçabilité** grâce aux événements (audit, logs)

• **Extensibilité** : ajouter de nouveaux consommateurs est simple

•**Résilience** (un service peut tomber sans bloquer tout le système)

Limites / défis

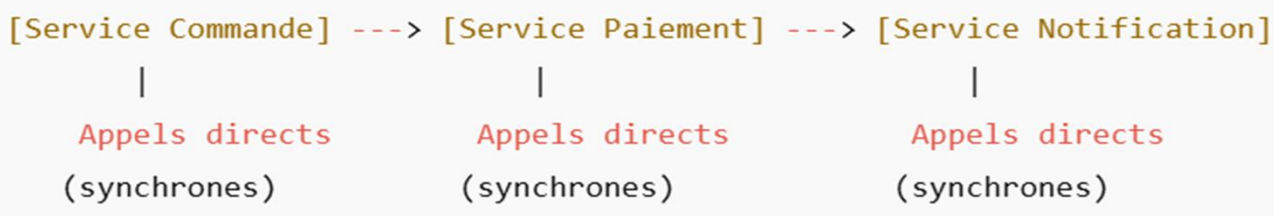
•Complexité du **suivi des événements** et du **debugging**

•Gestion des **échecs et de la résilience**

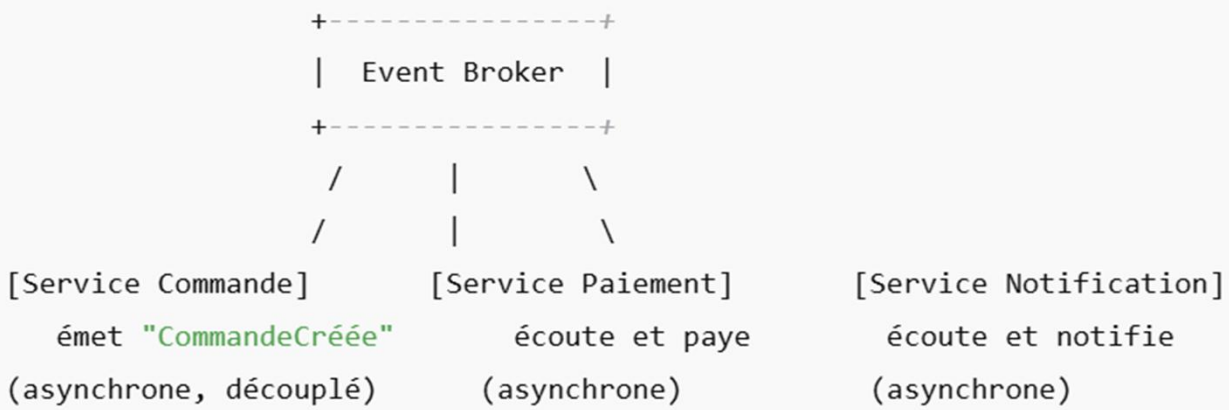
•Eventual consistency : les données ne sont pas toujours immédiatement cohérentes

Critère	SOA (Service-Oriented Architecture)	EDA (Event-Driven Architecture)
Communication	Synchronous (appel direct entre services via SOAP ou REST)	Asynchronous (services communiquent via des événements)
Couplage	Couplage plus fort : le service appelant doit connaître le service appelé	Couplage faible : les producteurs et consommateurs ne se connaissent pas
Réactivité	Réactif à la demande (pull)	Réactif aux changements (push)
Scalabilité	Scalabilité limitée, souvent liée aux services appelés	Scalabilité élevée, chaque service peut être mis à l'échelle indépendamment
Traçabilité / Suivi	Facile à tracer via les appels directs	Plus complexe, nécessite un Event Broker et monitoring des événements
Gestion de la cohérence	Cohérence forte et immédiate	Cohérence éventuelle (eventual consistency)
Exemple typique	Un service de commande appelle directement le service de paiement pour traiter la commande	Un service de commande émet `CommandeCréée`, et les services Paiement et Notification réagissent indépendamment
Complexité	Moins complexe à comprendre, mais plus rigide	Plus complexe à mettre en place et à déboguer, mais plus flexible
Adapté pour	Systèmes avec processus séquentiels et cohérence stricte	Systèmes réactifs, distribués et en temps réel, microservices à grande échelle

schéma visuel simple comparant SOA et EDA dans le contexte des microservices



- Chaque service **appelle directement** le suivant.
- Couplage fort : si un service est lent ou tombe en panne, tout le processus est impacté.
- Cohérence immédiate : chaque action attend la réponse du service appelé.



- Le service Commande **n'appelle pas directement** les autres services.
- Les services **écoutent les événements** qui les intéressent.
- Découplage fort, scalable, résilient : un service peut tomber sans bloquer les autres.
- Réactivité en temps réel et possibilité d'ajouter facilement de nouveaux consommateurs d'événements.

Comment fonctionne l'architecture orientée événements

Dans une **architecture orientée événements** :

•Les microservices **ne communiquent pas directement** entre eux.

•Ils échangent des **événements**, qui représentent un **changement d'état** ou une **action réalisée** dans le système.

•Chaque service peut être un **producteur** (il émet des événements) ou un **consommateur** (il réagit aux événements), ou les deux.

Exemple d'événement
•CommandeCréée → Une nouvelle commande a été enregistrée
•PaiementEffectué → Une commande a été payée
•UtilisateurInscrit → Un nouvel utilisateur est enregistré

Les composants clés

1.Microservice producteur

1. Ex : Service Commande
2. Rôle : émettre un événement lorsqu'une action est faite (ex. une commande est créée)

2.Event Broker (courtier d'événements)

1. Ex : Kafka, RabbitMQ
2. Rôle : transporter les événements aux microservices intéressés
3. Avantage : découplage total entre producteurs et consommateurs

3.Microservice consommateur

1. Ex : Service Paiement, Service Notification
2. Rôle : écouter les événements et réagir en conséquence (payer la commande, envoyer un mail)

Fonctionnement étape par étape

Étape 1 : Une commande est créée

•Le **Service Commande** enregistre la commande dans sa base de données.

•Il **émet un événement** CommandeCréée au broker.

Étape 2 : Les services abonnés réagissent

•**Service Paiement** : écoute CommandeCréée → effectue le paiement → émet éventuellement PaiementEffectué.

•**Service Notification** : écoute CommandeCréée → envoie un email ou notification au client.

Étape 3 : Découplage et asynchronisme

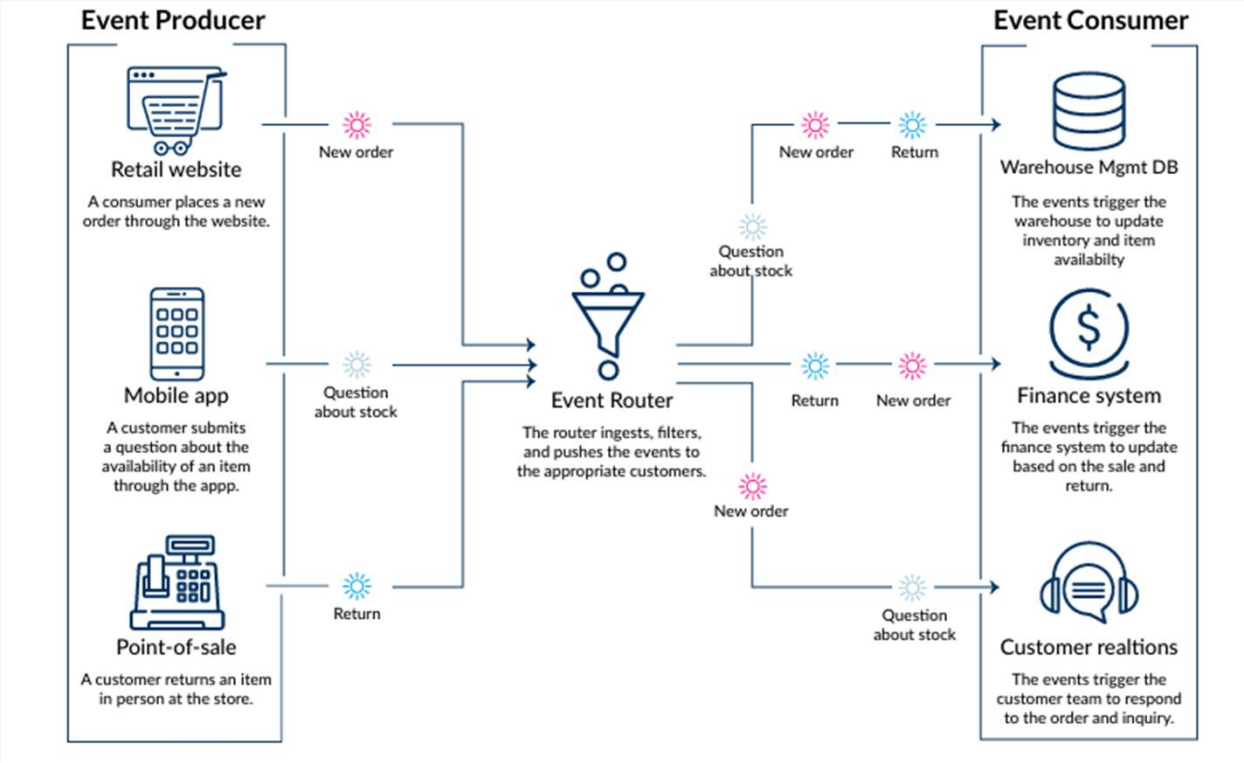
•Aucun service n'appelle directement un autre service.

•Chaque microservice est **indépendant** : il peut être développé, testé et mis à l'échelle seul.

•Les événements assurent la **coordination** et le **suivi des actions**.

Exemple

Ci-dessous, un **exemple d'un site e-commerce** utilisant une architecture orientée événements :
Pendant les périodes de forte demande, cette architecture permet au site de **réagir aux changements provenant de multiples sources** sans **planification excessive des ressources** ni **risque de plantage**.



Modèles d’architecture orientée événements

Un **modèle d’architecture orientée événements** décrit **comment les événements sont produits, transmis et consommés** dans un système.

👉 Tous les modèles EDA reposent sur la même idée :

Un événement se produit → il est publié → des composants réagissent.

Mais **la façon de distribuer et traiter ces événements** varie selon le modèle

Les principaux modèles d'architecture orientée événements: Modèle Publisher / Subscriber (Pub/Sub)

Le modèle **Publisher/Subscriber** (ou **Pub/Sub**) est un **patron d'architecture orienté événements** dans lequel :

•**Publishers (éditeurs)** : produisent des événements ou des messages.

•**Subscribers (abonnés)** : consomment les événements ou messages qui les intéressent.

•**Broker / Middleware (intermédiaire)** : optionnel, il reçoit les messages des publishers et les transmet aux subscribers correspondants.

L'idée clé : **les publishers et subscribers sont découplés**. Le publisher ne sait pas qui consomme ses messages, et le subscriber ne sait pas qui produit les messages.

Fonctionnement étape par étape

1.Le **publisher** envoie un message ou un événement au **broker**.
2.Le **broker** identifie tous les **subscribers** intéressés par ce type d'événement (souvent grâce à un **topic** ou un **channel**).
3.Le **broker** distribue le message à tous les subscribers inscrits.

Concepts importants

Concept	Description
Publisher	Génère des événements ou messages.
Subscriber	Reçoit et traite les événements.
Topic / Channel	Catégorie ou canal qui classe les messages.
Broker	Système qui reçoit les messages et les distribue (Kafka, RabbitMQ, etc.).
Découplage	Publisher et Subscriber ne se connaissent pas directement.
Asynchrone	Les messages sont souvent traités de façon asynchrone.

Exemple concret

Imaginons un **site e-commerce** :

1.Lorsqu'un client **passe une commande** :

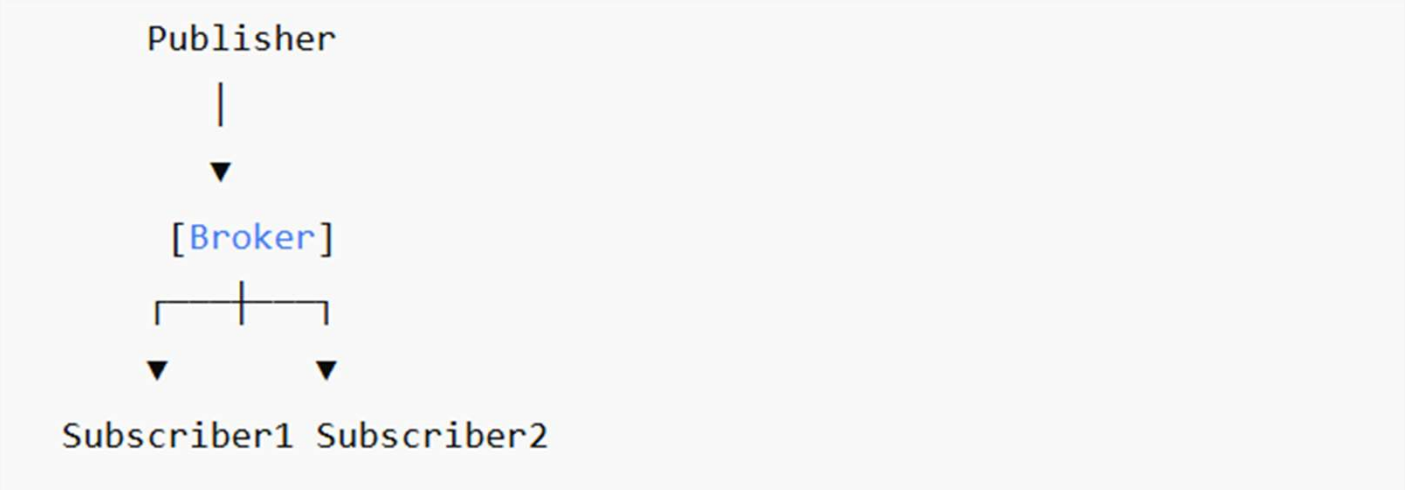
1. Publisher : le service **Commande** publie l'événement CommandeCréée.

2.Subscribers possibles :

1. Service **Facturation** → génère la facture.
2. Service **Stock** → met à jour les quantités.
3. Service **Notification** → envoie un email au client.

Chaque service réagit indépendamment et en parallèle, sans que le service Commande sache qui les consomme.

Schéma simplifié ASCII



- Les flèches montrent la **diffusion des messages**.
- Le broker assure le **découplage et la distribution**.

Avantages

- Découplage fort entre producteurs et consommateurs.
- Haute scalabilité : on peut ajouter autant de subscribers que nécessaire.
- Flexibilité et maintenance simplifiée : ajouter un nouveau subscriber **n'affecte pas les publishers existants**.
- Asynchrone : les publishers ne sont pas bloqués en attendant que les subscribers traitent les messages.

Le Pub/Sub est idéal pour des architectures **évolutives, asynchrones et distribuées**, très utilisé dans les **microservices** ou les systèmes nécessitant une **réaction en temps réel aux événements**.

Modèle Event Streaming

Le **Event Streaming** est un modèle où **les événements sont publiés et stockés dans un flux continu** (stream), et peuvent être **consommés en temps réel ou relus plus tard**.
Contrairement au Pub/Sub classique :
•Dans Pub/Sub, un message est **transmis et disparaît** après consommation.
•Dans Event Streaming, les événements sont **persistés dans un journal (log)** et peuvent être **rejoués ou consommés par plusieurs services à des moments différents**.

Fonctionnement étape par étape
1.Un **producteur (producer)** envoie des événements vers un **stream**.
2.Les événements sont **persistés dans un log ordonné**.
3.Les **consommateurs (consumers)** lisent les événements à leur rythme, indépendamment.
4.Les consommateurs peuvent **revenir en arrière** pour relire des événements passés si nécessaire.

Concepts clés

Concept	Description
Producer	Produit des événements et les écrit dans un stream.
Consumer	Lit les événements du stream, souvent à son propre rythme.
Stream / Log	Journal ordonné des événements, persistant dans le temps.
Partition	Un stream peut être découpé pour scaler horizontalement.
Offset	Position d'un événement dans le stream, utilisé pour reprendre la lecture.
Durabilité	Les événements restent stockés et peuvent être rejoués.

Exemple concret

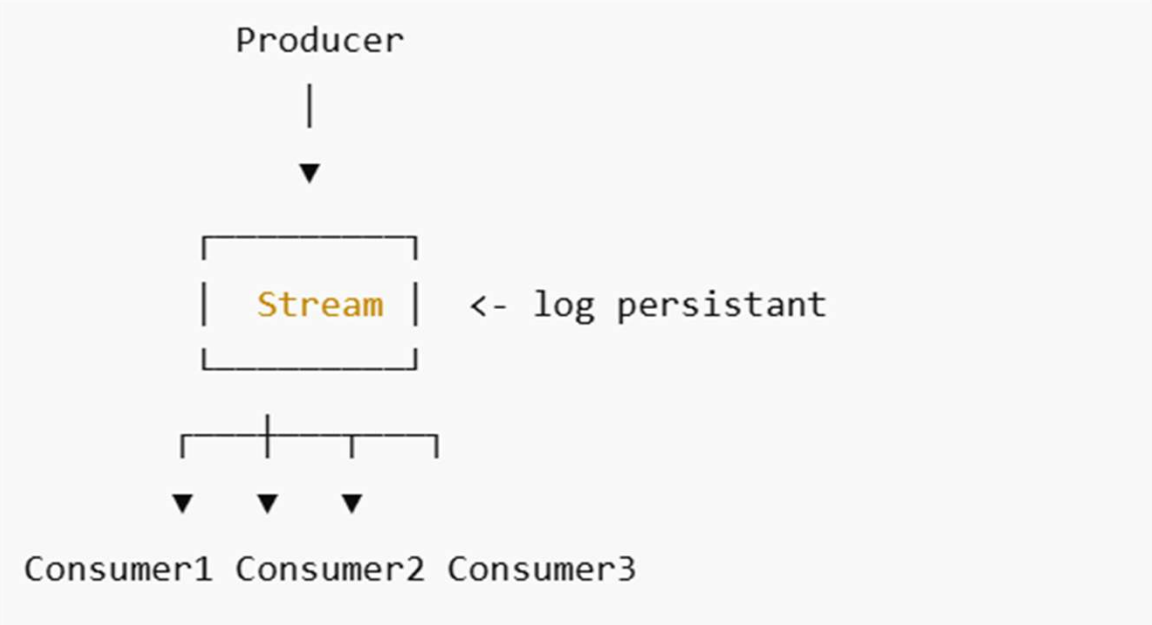
Prenons le **même site e-commerce** :

1. Quand un client **passe une commande** :
    1. Producer : le service **Commande** envoie l'événement CommandeCréée vers le **stream de commandes**.
2. Consumers :
    1. Service **Facturation** lit le stream pour générer la facture.
    2. Service **Stock** lit le stream pour mettre à jour le stock.
    3. Service **Notification** lit le stream pour envoyer un email.
3. Nouvel arrivant :
    1. Si demain un service **Analytics** est ajouté, il peut **rejouer le stream complet** pour analyser toutes les commandes passées.

La différence clé avec Pub/Sub : ici, **les événements sont stockés**, pas seulement transmis.

Les consumers peuvent lire à différents moments.

Les événements restent dans le stream jusqu’à expiration ou suppression.



Avantages

Persistance des événements : possible de rejouer les événements.
Découplage fort : producers et consumers évoluent indépendamment.
Scalabilité horizontale : les partitions permettent de distribuer la charge.
Idéal pour les systèmes **réactifs en temps réel**, analytics, monitoring.

Le Event Streaming est une évolution du Pub/Sub pour les architectures **data-driven**, où les événements doivent être **persistés, rejouables et scalables**, comme avec **Apache Kafka**, **Redpanda** ou **AWS Kinesis**.

Modèle Event Sourcing

L'**Event Sourcing** est un modèle d'architecture où **l'état d'une application n'est pas stocké directement**, mais **reconstruit à partir d'une séquence d'événements passés**.

•Chaque **changement d'état** est représenté par un **événement**.

•L'état actuel est obtenu en **rejouant tous les événements** depuis le début.

L'idée clé : **les événements deviennent la source unique de vérité** (single source of truth).

Fonctionnement étape par étape

1.Un utilisateur effectue une action (ex. : créer un compte ou passer une commande).

2.Cette action génère un **événement** (ex. : CompteCréé, CommandePayée).

3.L'événement est **persisté dans un journal** (Event Store).

4.Pour connaître l'état actuel d'un objet (ex. un compte bancaire), on **rejoue tous ses événements** dans l'ordre.

Concept	Description
Event	Représente une modification de l'état.
Event Store	Journal persistant de tous les événements.
Aggregate	Objet métier dont l'état est reconstruit en appliquant les événements.
Replay	Relecture des événements pour reconstruire l'état actuel.
Immutabilité	Les événements sont **immuables**, on ne les modifie jamais.

Concept	Description
Event	Représente une modification de l'état.
Event Store	Journal persistant de tous les événements.
Aggregate	Objet métier dont l'état est reconstruit en appliquant les événements.
Replay	Relecture des événements pour reconstruire l'état actuel.
Immutabilité	Les événements sont **immuables**, on ne les modifie jamais.

Exemple : compte bancaire

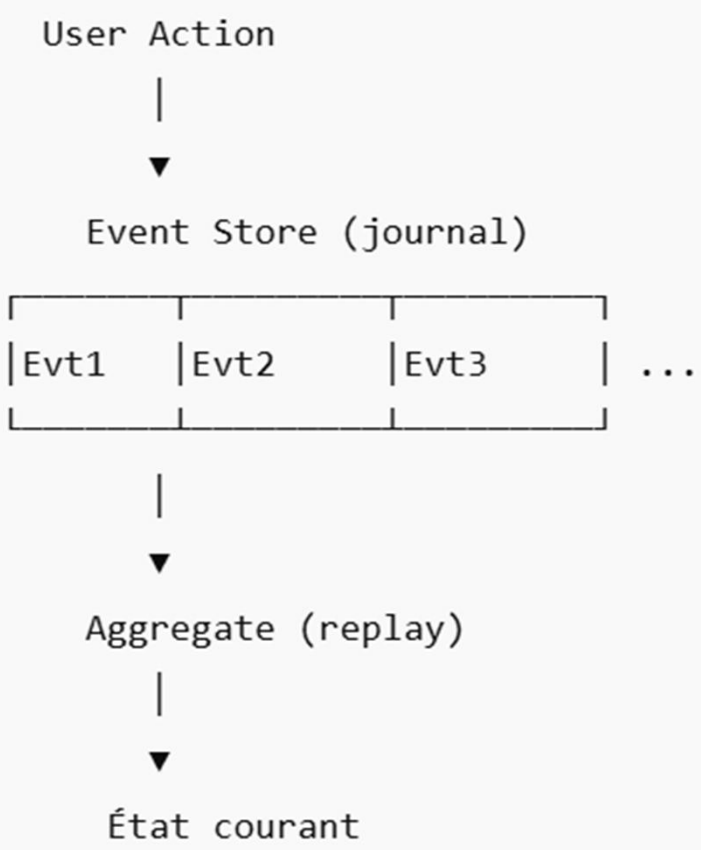
1. Actions de l'utilisateur :
    1. CompteCréé
    2. DépotEffectué(100)
    3. RetraitEffectué(30)
2. Event Store :

[CompteCréé, DépotEffectué(100), RetraitEffectué(30)]

3. État actuel du compte :
    3. On **rejoue** les événements :
        3. solde initial = 0
        4. après dépôt = 100
        5. après retrait = 70

L'état actuel (70) est **le résultat des événements**, pas un simple champ mis à jour dans une table.

Schéma simplifié ASCII



Avantages

- **Audit complet** : chaque changement est historisé.
- **Reconstruction de l'état** à tout moment.
- Permet de **rejouer des événements** pour corriger des bugs ou refaire des calculs.
- Excellent pour les systèmes distribués, microservices et CQRS.

'Event Sourcing est idéal pour les systèmes où **l'historique complet des actions est important**, où les états peuvent être **reconstitués à tout moment**, et souvent combiné avec **Event Streaming** pour diffuser ces événements aux autres services.

Modèle CQRS + EDA

•**CQRS (Command Query Responsibility Segregation)** :

C'est un modèle où **la lecture (Query)** et **l'écriture (Command)** des données sont **séparées**.

- Command : modifie l'état (ex. créer ou mettre à jour un compte).
- Query : lit l'état actuel (ex. afficher le solde).

•**EDA (Event-Driven Architecture)** :

Les **modifications d'état sont représentées par des événements**, qui sont publiés et consommés par différents services.

CQRS + EDA combine les deux :

•Les **Commands** génèrent des **events**.

•Les **events** mettent à jour **les modèles de lecture** (Read Models) et déclenchent des actions dans d'autres services.

L'idée clé : **séparer les lectures et les écritures et propager les changements via des événements**.

Fonctionnement étape par étape

1. L'utilisateur envoie une **commande** (Command) à un service.
    1. Exemple : RetirerArgent(100)
2. Le service valide la commande et publie un **événement** : ArgentRetire(100).
3. L'événement est envoyé via un **broker** (Kafka, RabbitMQ…).
4. Les **Read Models** et autres services abonnés (Subscribers) reçoivent l'événement et mettent à jour leur état ou déclenchent des actions.
5. Les utilisateurs peuvent maintenant **lire l'état à jour** depuis les Read Models optimisées pour la lecture.

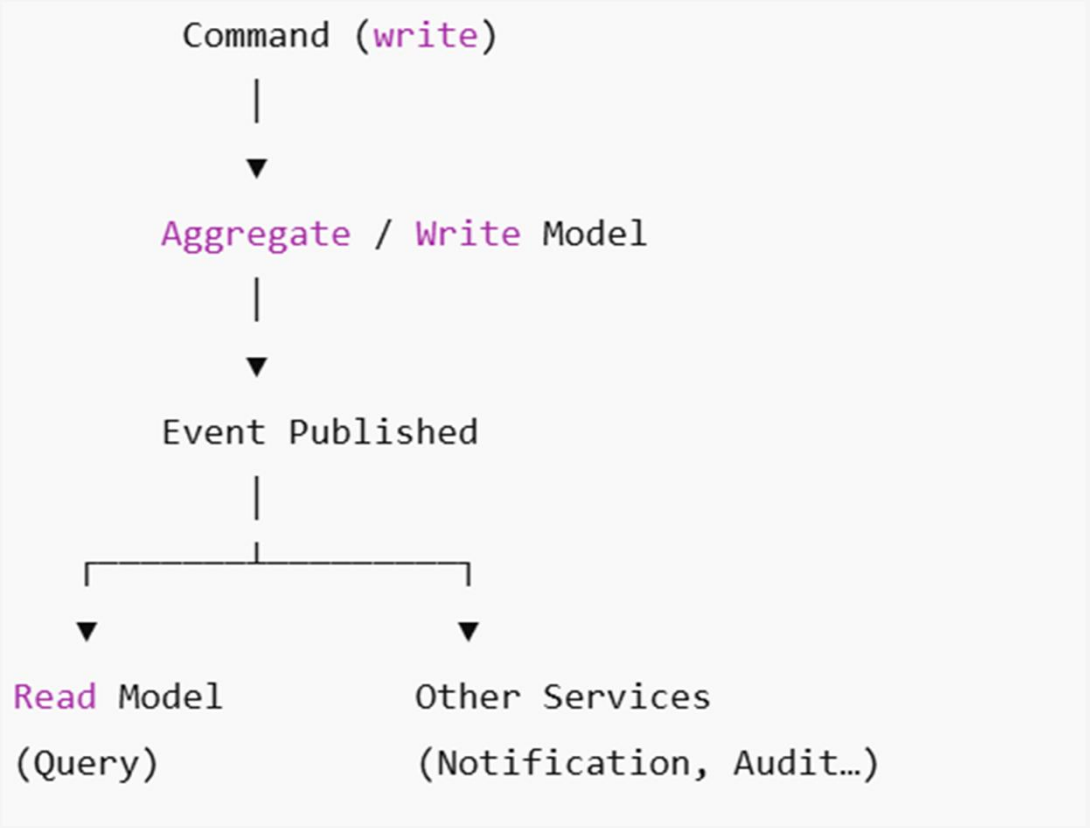
Concepts clés

Concept	Description
Command	Action qui **modifie l'état**.
Event	Notification d'un changement d'état.
Read Model / Query	Représentation optimisée pour la **lecture**.
Write Model / Aggregate	Représentation pour **écrire et appliquer les changements**.
Broker / Event Bus	Transporte les événements vers tous les services intéressés.
Découplage	Les services producteurs et consommateurs ne se connaissent pas directement.

Exemple concret : Compte bancaire

1. Commande : RetirerArgent(50)
2. Service Compte → publie ArgentRetire(50)
3. Subscribers :
    1. Read Model → met à jour le solde du compte pour la lecture.
    2. Notification Service → envoie un SMS au client.
    3. Audit Service → enregistre l'événement pour l'historique.

Séparation claire : la **logique d'écriture** est dans l'Aggregate, la **logique de lecture** est dans les Read Models.



Les **Read Models** sont séparées du **Write Model**.
Les **events** propagent les changements à tous les services concernés.

Avantages

•**Séparation claire** lecture/écriture → performance et scalabilité améliorées.

•Propagation asynchrone des événements → **découplage** entre services.

•Historique complet des événements → **audit et reconstruction** possibles.

•Adapté aux **microservices et architectures distribuées**

CQRS + EDA = architecture moderne pour **systèmes distribués et microservices**, combinant :

•Commands → modification d'état

•Events → propagation des changements

•Read Models → lecture optimisée et indépendante

Modèle Event-Driven Choreography

Event-Driven Choreography est un modèle dans lequel **plusieurs services interagissent uniquement via des événements**, **sans orchestrateur central**.

•Chaque service est **autonome** et réagit aux événements qu'il reçoit.

•Les services **ne savent pas qui déclenche leurs actions**.

•La coordination globale émerge **naturellement** à partir des événements, d'où le terme **"choreography"** (chorégraphie).

L'idée clé : **aucun contrôle central**, tout se passe par les événements qui circulent dans le système.

Fonctionnement étape par étape

1.Un service publie un **événement** quand quelque chose change (ex. : CommandeCréée).

2.D'autres services abonnés reçoivent cet événement et réagissent (ex. : préparer la livraison, mettre à jour le stock, envoyer un email).

3.Chaque service peut **émettre à son tour de nouveaux événements**, qui déclenchent d'autres actions.

4.Ainsi, une **choregraphie de services** se met en place **via le flux d'événements**.

Concepts clés

Concept	Description
Service autonome	Chaque service prend ses décisions en fonction des événements reçus.
Event	Déclencheur d'action ou notification de changement.
Broker / Event Bus	Transporte les événements à tous les services abonnés.
Choreography	Coordination décentralisée via les événements, pas d'orchestrateur.
Découplage	Les services ne se connaissent pas directement.

Exemple concret : Site e-commerce

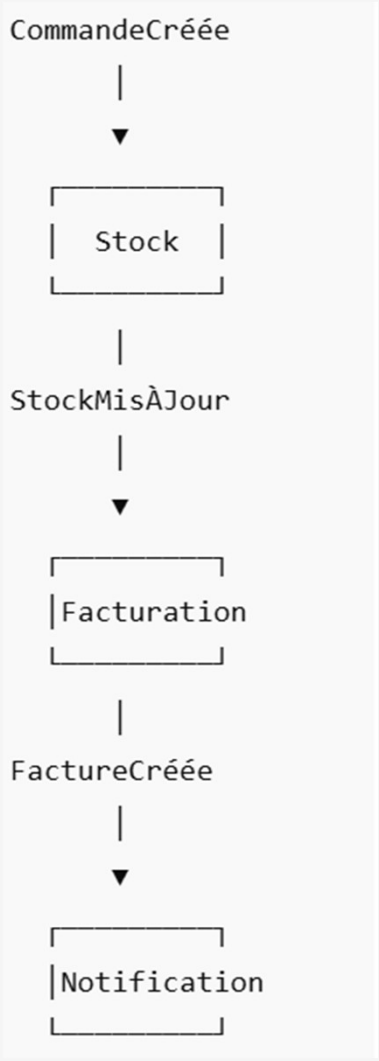
1. **Service Commande** : publie CommandeCréée.
2. **Service Stock** : reçoit l'événement → réserve les produits → publie StockMisÀJour.
3. **Service Facturation** : reçoit CommandeCréée → génère la facture → publie FactureCréée.
4. **Service Notification** : reçoit FactureCréée ou StockMisÀJour → envoie un email au client.

Chaque service agit **indépendamment**, la coordination globale se fait **via les événements**.

Schéma simplifié ASCII

Chaque service **publie et consomme** des événements.

Pas de contrôleur central, tout est basé sur la **choregraphie**.



Avantages

- **Découplage fort** entre services.
- Haute **scalabilité** et résilience (pas de point unique de défaillance).
- Facilité d'évolution : on peut ajouter de nouveaux services qui réagissent aux événements existants.
- Architecture naturelle pour les systèmes **réactifs et distribués**.

L'Event-Driven Choreography = orchestration **décentralisée**, où **chaque service joue sa partition** et la coordination globale **émerge naturellement** via les événements.

Modèle	Objectif / Utilité	Communication	événements	Découplage	Scalabilité	Exemples / Cas d'usage
Pub/Sub	Diffuser des messages à plusieurs consommateurs	Asynchrone	Non persistant	Fort	Élevée	Notifications, messagerie, microservices
Event Streaming	Flux d'événements persistants et rejouables	Asynchrone	Persistant	Fort	Très élevée	Kafka, Kinesis, analytics temps réel
Event Sourcing	Stocker l'historique complet des changements d'état	Asynchrone	Persistant	Moyen	Moyenne	Comptes bancaires, audit, systèmes financiers
CQRS + EDA	Séparer lecture et écriture et propager via événements	Asynchrone	Persistant/Optionnel	Fort	Très élevée	Microservices distribués, e-commerce
Event-Driven Choreography	Coordination décentralisée via événements, sans contrôleur	Asynchrone	Persistant/Optionnel	Très fort	Très élevée	E-commerce, systèmes réactifs, IoT

Quand utiliser une architecture orientée événements ?

1. Réplication des données entre comptes et régions
Grâce à une architecture orientée événements, les systèmes peuvent être **coordonnés entre des équipes travaillant et déployant sur plusieurs sites et comptes**.
Vous pouvez **créer, dimensionner et déployer les services de manière indépendante** en utilisant un **routeur d'événements** pour transporter les données entre les systèmes.
2. Traitement parallèle et fanout
Avec une architecture orientée événements, vous pouvez **diffuser un événement à plusieurs systèmes** en réponse à cet événement, sans avoir à écrire du code pour chaque consommateur.
Les systèmes recevront l'événement via le routeur, qu'ils pourront ensuite **traiter en parallèle** pour différents objectifs.
3. Alerte et surveillance de l'état des ressources
Vous pouvez utiliser une architecture orientée événements pour **surveiller et recevoir des notifications** sur toute **anomalie, changement ou mise à jour**, plutôt que de vérifier continuellement vos ressources.
Ces ressources peuvent inclure des **nœuds de calcul, des fonctions serverless, des tables de bases de données, des buckets de stockage**, et plus encore.
4. Mise en relation de systèmes divers
Une architecture orientée événements peut être utilisée pour **transmettre des informations entre des systèmes fonctionnant sur des stacks différents**, sans couplage.
Grâce à la **mise en place d'indirection et d'interopérabilité par le routeur d'événements**, les systèmes peuvent **échanger des messages et des données tout en restant indépendants**.

Définition du Broker

Un **Broker** est un **intermédiaire qui reçoit, stocke et distribue des messages ou événements** entre producteurs (publishers/producers) et consommateurs (subscribers/consumers).

•Il **découple** totalement les producteurs et les consommateurs.

•Il peut gérer la **persistabilité, la livraison garantie et le routage des messages**.

On peut le voir comme **le "poste central" des messages** dans un système distribué

Fonctionnement du Broker

1.Le **producteur** envoie un message ou un événement au broker.
2.Le **broker** stocke éventuellement le message (persistant ou temporaire).
3.Le broker **routage le message** vers les consommateurs abonnés (subscribers).
4.Les consommateurs reçoivent le message **à leur rythme**, sans bloquer le producteur.

Concepts clés du Broker

Concept	Description
Topic / Channel	Catégorie ou canal pour publier/consommer les messages.
Queue	File d'attente de messages pour les consommateurs.
Delivery Guarantee	Garantie que le message sera livré au moins une fois (at-least-once) ou exactement une fois (exactly-once).
Persistence	Les messages peuvent être stockés pour rejouer ou récupérer en cas de panne.
Routing / Filtering	Le broker peut envoyer le message seulement aux consommateurs concernés.

Exemples de Brokers connus

Broker	Type	Caractéristique principale
Apache Kafka	Event Streaming	Très performant, persistance et partitions
RabbitMQ	Pub/Sub / Queue	Supporte routing avancé et persistance optionnelle
ActiveMQ	Pub/Sub / Queue	Similaire à RabbitMQ
AWS SNS/SQS	Cloud Pub/Sub	Pub/Sub géré, scalable
Redpanda	Event Streaming	Kafka compatible, haute performance

Le **broker** est le cœur des architectures orientées événements : il **assure la communication fiable et asynchrone** entre tous les services, sans qu'ils aient besoin de se connaître.