Smile

Utiliser Kafka

Cible et pré-requis

- Débutants
- Utilisation de Kafka avec Java

- Linux
- Docker
- Docker Compose
- Java 17

Apache Kafka, qu’est-ce donc ?

- Message Oriented Middleware (MOM)
  - Intergiciel orienté message
  - Logiciel qui va échanger des messages entre vos différentes briques

- Depuis 2011
  - Stable
  - Largement utilisé
  - Utilisé par 80% du fortune 100

- Open source
  - Fondation Apache

Apache Kafka, pourquoi donc ?

- Haute dispo
  - Tolérance aux pannes

- Evolutivité (scalability)
  - Permet de gérer facilement de forts écarts de trafic
  - Économies en temps de faible trafic

- Architecture guidée par les événements (Event Driven), Micro-services
  - Événements fonctionnels
  - Facilement adaptable aux changements

Plan

- Mise en place

- Concepts de base
  - Topic & Queue
  - Producer (exo)
  - Consumer (exo)
  - Message
  - Cluster
  - Replication
  - Schema-registry

- Kafka et Java
  - Spring-kafka

Mise en place

- Récupérer Kafka : https://kafka.apache.org/downloads
  - Mettre les bin dans le PATH

- Projet de la formation : https://git.smile.fr/mater/kafka

Queue ou Topic ?

- Les deux sont des files de messages
  - X producers
  - Y consumers

- Queue : 1 msg n’est lu qu’une fois, 1er arrivé 1er servi

- Topic : broadcast, tout le monde reçoit tous les msg

Et pour Kafka ?

- TL;DR : Les deux
  - On ne parle que de topic

- Beaucoup de souplesse
  - Partitions
  - Consumer group

Messages (aussi appelés Records)

- Payload
- Timestamp
- Partition key
  - Manuel pour contrôler la parallélisation
  - Automatique par défaut
- Offset
  - Outil de gestion de file par Kafka
  - Possible de lire et d’écrire
    - Pour du traitement de batch volumineux

Producer

- Publie des msg sur un topic
  - Authentification
  - Commit

- Spécifie le parallélisme des messages
  - Clé de partition

Exercice : envoyer un msg sur un topic et le visualiser avec akhq

Consumer

- Se connecte pour lire les msg (Kafka ne les pousse pas)
- Gère
  - Authentification
  - Acknowledgement
  - Poll interval
  - Batch size
- Consumer group id

Exercice : Créer un consumer sur le topic

Commit & Acknowledgement

Nomenclature

- Le producteur commit
- La consommateur acknowledge

Signifie la clôture de la transaction et le déplacement de l’offset.

! : Attention aux configurations par défaut qui ack à la lecture du msg et pas à la fin du traitement

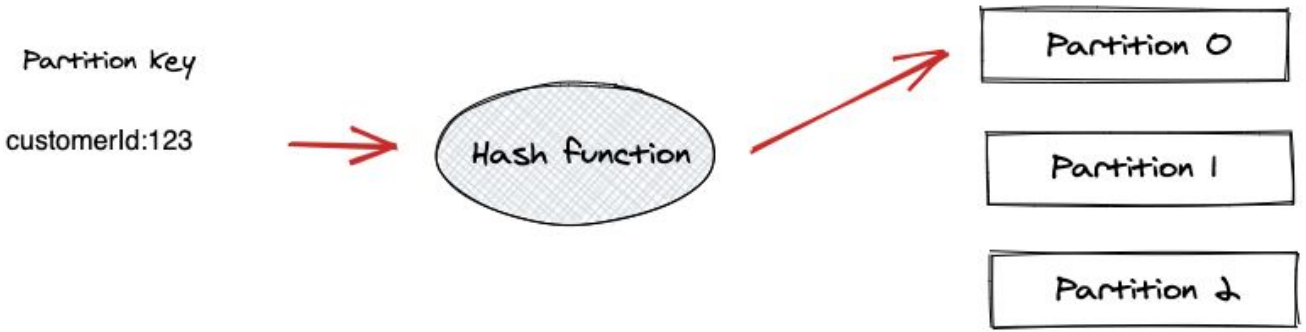
Mais en vrai, c’est quoi un topic ?

- Nom
- Nb de Partitions
  - Défaut = 1
- Facteur de Réplication
  - Nombre de fois que le topic est présent dans le cluster
  - Défaut = 1

Partitions

Segmentation d’un topic

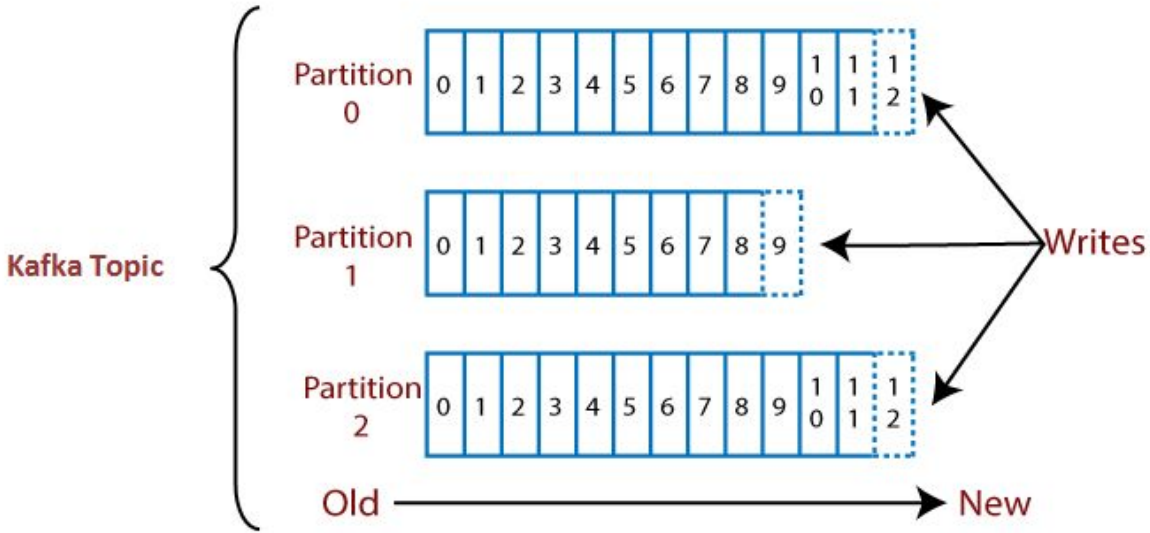
- Gérée à l’aide d’une fonction de hachage sur la “clé de partition” du msg



Partitions (avec un dessin)

Sert à assurer le traitement séquentiel des msg selon leurs clés

- Producer
  - Msg1 key=k1
  - Msg2 key=k2
  - Msg3 key=k1
- Partition1
  - Msg1
  - Msg3
- Partition2
  - Msg2



Partition & Consumer group

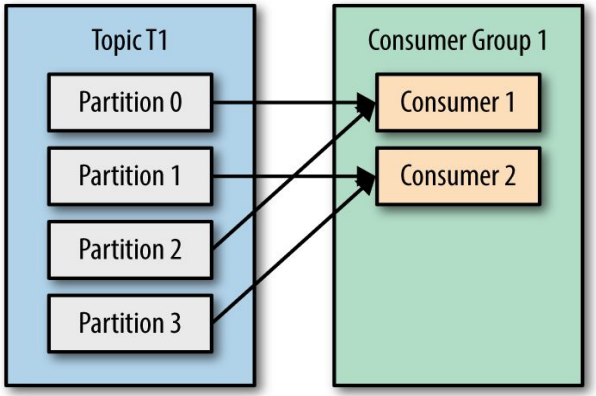
- Permet
  - Le traitement parallèle des msg
    - 2 partitions sont lues en même temps par les consumers d’un même groupe
  - Le traitement séquentiel des msg
    - Chaque partition n’est lue que par 1 consumer par groupe

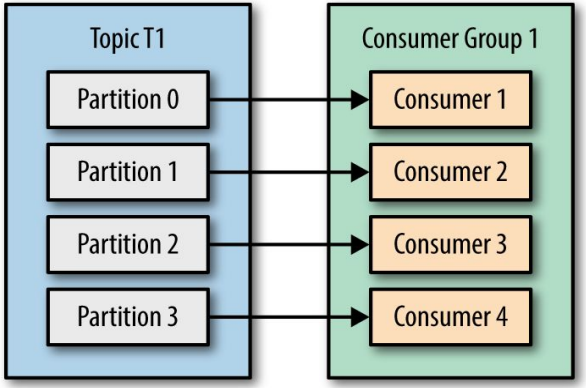
- Consumer group id identiques
  - Un seul consumer reçoit le msg (queue)
  - Cas d'utilisation: Plusieurs instances du même service

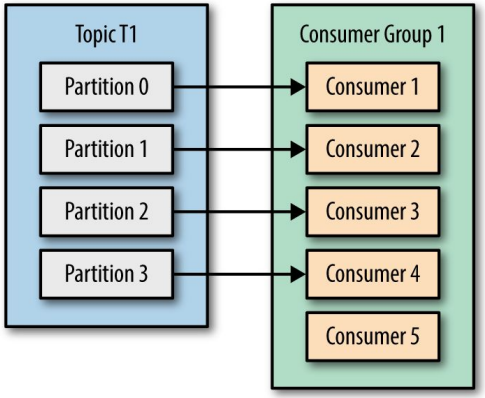
- Consumer group id différents
  - Tous les consumers reçoivent le msg (topic)
  - Cas d'utilisation: des services différents écoutent le même événement

Exercice : Créer un topic avec 2 partitions ainsi que 2 consumers dans le même groupe sur ce topic. Envoyer des msg avec différentes clés de partitions

Consommateurs et Partitions







Consommateurs et Partitions, implications

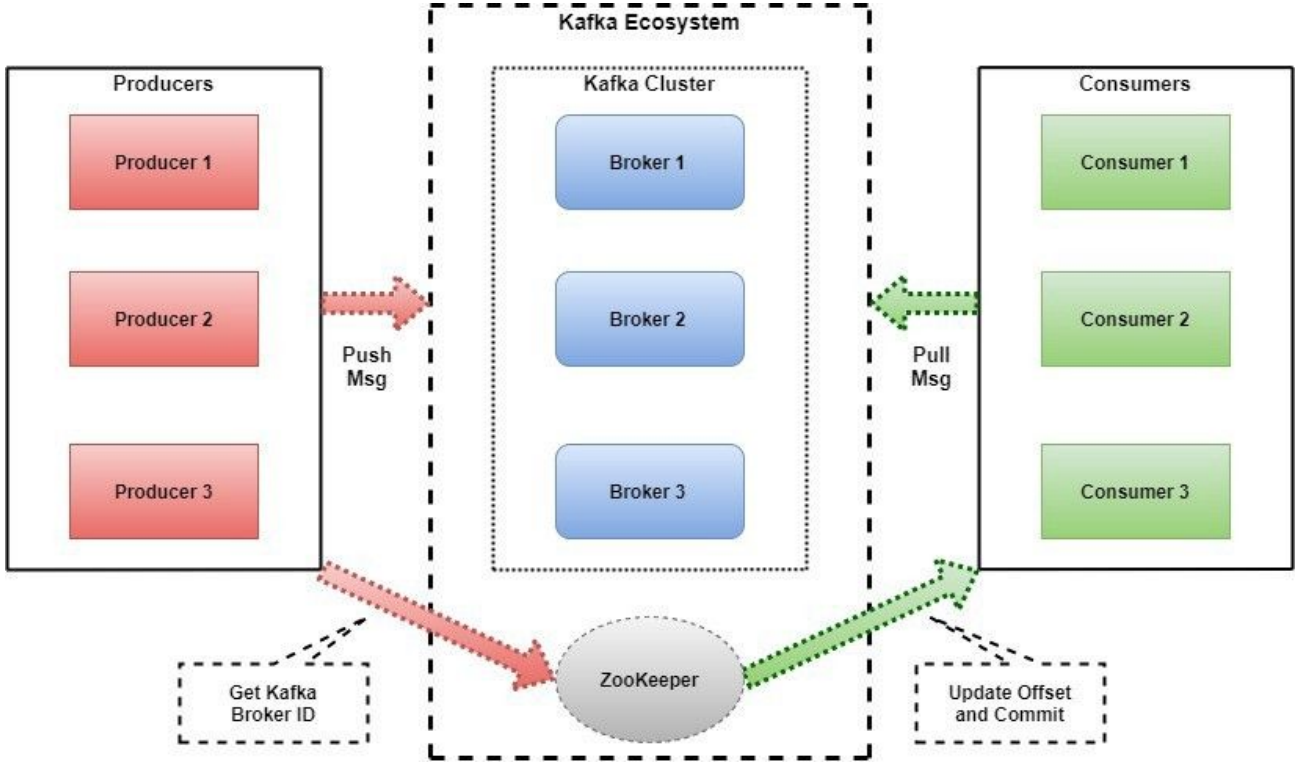
- Ni trop
  - Plus de réplication => plus de latence pour synchro
  - Plus de partitions => plus de coût d'I/O sur les brokers

- Ni trop peu
  - Difficile à augmenter pendant le run
    - Pas infaisable
  - Prévoir la croissance

- Pas de réponse précise
  - Dépend du trafic et de sa croissance attendue

Kafka en vrai, c’est quoi ?

- Kafka = cluster
- Broker = noeud
- ZooKeeper
  - Organise le bordel
    - Conf
    - Naming
    - Synchro
    - …



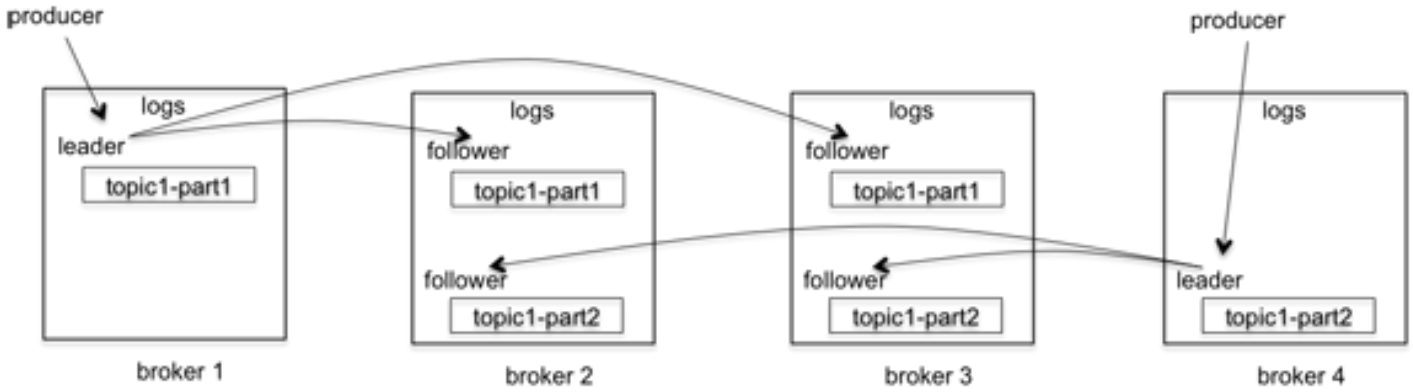
Dans un broker

- Messages dans des logs sur le disque
  - Espace réservé en mémoire
  - Un log par topic
  - Non c’est pas lent
    - Write Ahead Log

- Broker Leader sur une partition sur un topic
  - Les msg sont redirigés sur le leader
  - Les msg sont ensuite dupliqués sur les autres brokers Followers

Dessine moi une réplication

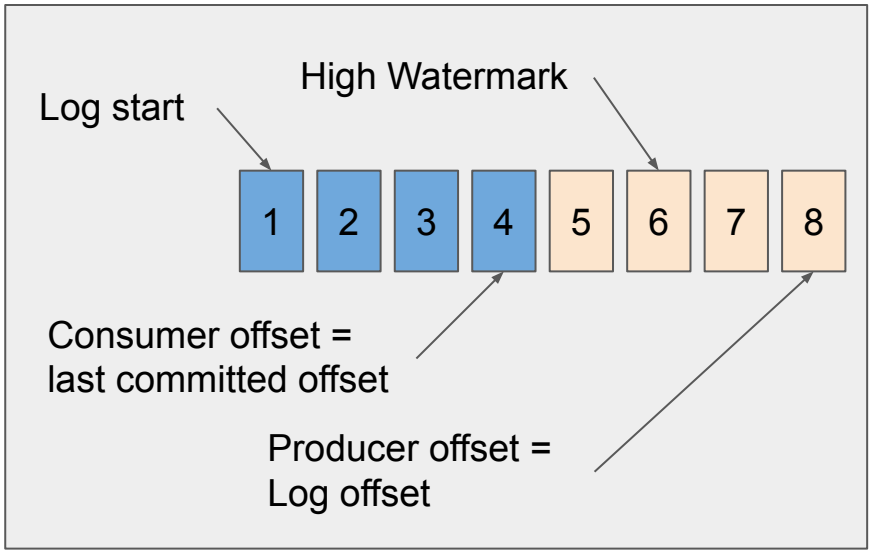
4 broker, 1 topic, 2 partitions, replication factor = 3

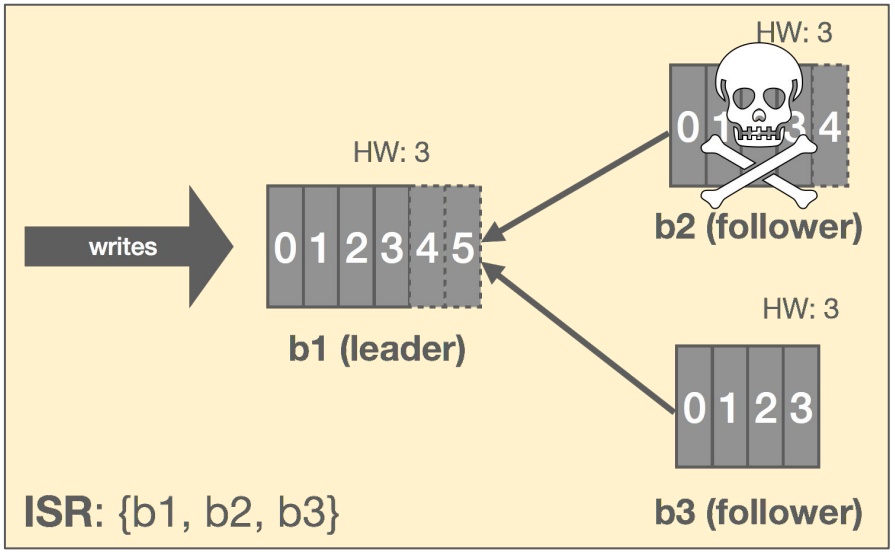


Les leaders sont distribués uniformément sur les brokers

Connexion de %*$! Ca lag

- Consumer lag
  - Différence entre offset du dernier msg et l'offset du dernier msg lu
- Replication lag
  - Différence d'offset entre réplicas





Pour aller plus loin

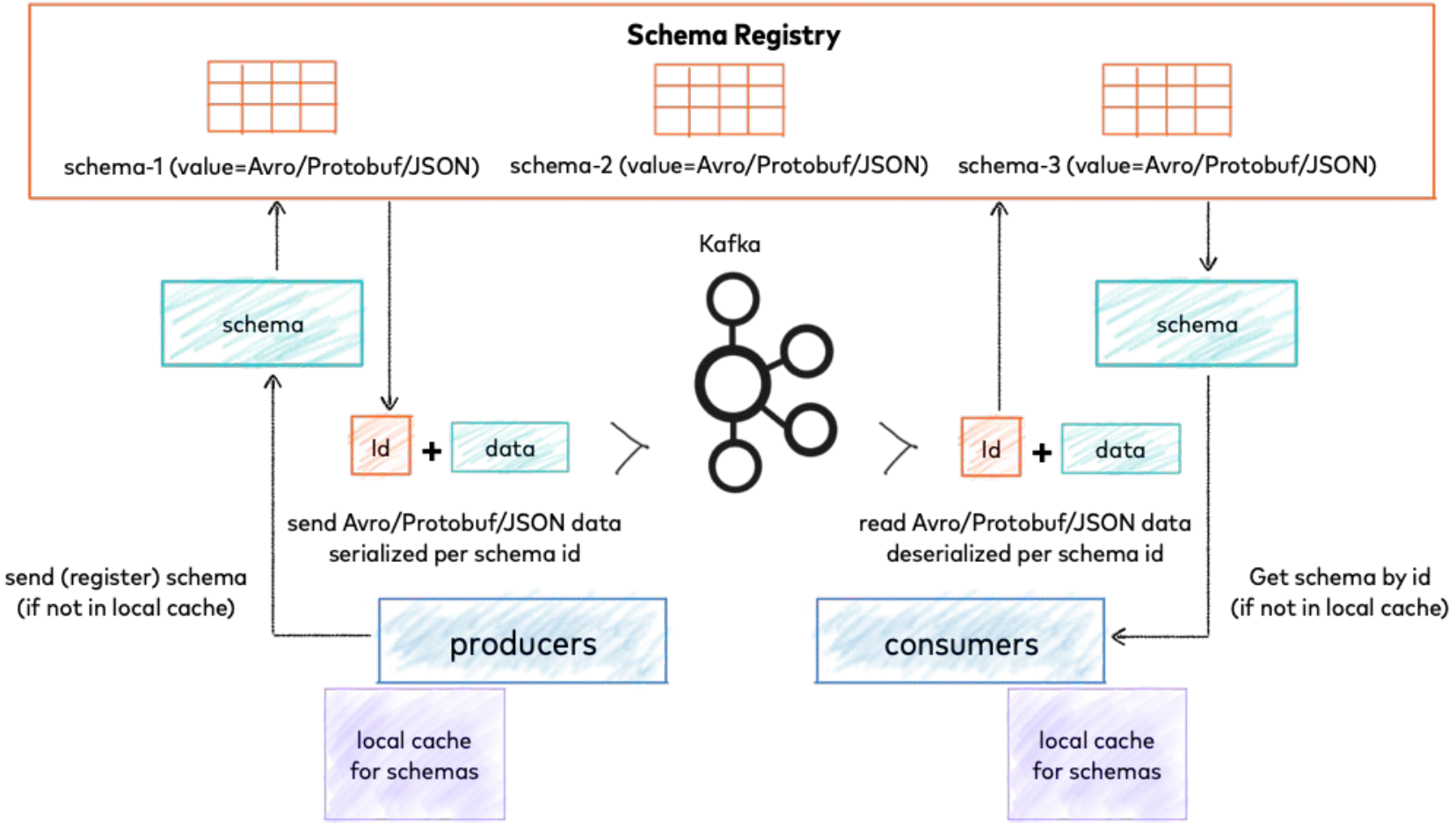
- In-Sync Replica list (ISR)
  - Replica.lag.max.messages
  - Replica.lag.time.max.ms

- Election de leader
  - Preferred replica election
    - Attention aux leaders “unclean” (pas dans l’ISR)
  - Premier arrivé (présent de l’ISR)

- ZooKeeper aussi doit être répliqué (et donc également élection de leader)
  - Corum

Data validation avec Schema registry

- Centralise et historise les schémas des données par topic
- Distribué
- Réutilise Kafka
  - Persistance (Write Ahead Log)
  - Gestion de la distribution (ZooKeeper)
  - Election de leader (Kafka)


Schema Registry
schema-1 (value=Avro/Protobuf/JSON)
schema-2 (value=Avro/Protobuf/JSON)
schema-3 (value=Avro/Protobuf/JSON)
Kafka
schema
schema
Id
+
data
Id
+
data
send Avro/Protobuf/JSON data
serialized per schema id
read Avro/Protobuf/JSON data
deserialized per schema id
send (register) schema
(if not in local cache)
Get schema by id
(if not in local cache)
producers
consumers
local cache
for schemas
local cache
for schemas

Comment ça marche ?

Producer :

- POST schema vers Schema Registry, récupère un id de schéma
- POST msg + id vers Kafka

Consumer :

- GET msg + id
- GET schema

La validation est à charge du producteur et du consommateur.

Exercice : POST et GET un schéma sur le schema registry

Kafka et Java (Spring Kafka)

- Producer
  - KafkaTemplate.sendMessage()
  - Bean topic

- Listener
  - @KafkaListener(id = "myId", topics = "topic1")

Exo Java + schema registry

1. Publier un msg avec et sans id de schéma
    a. Lancer un consumer console
    b. Lancer le producer Java et un producer console
    c. Publier un msg avec le producer Java & le producer console
    d. Voir le schéma dans le schema registry
2. Lire un msg avec schema valide
    a. Lancer les consumer & producer Java
    b. Publier un msg avec le producer Java
3. Changer le schéma dans le schema registry
    a. Voir les compatibilités : https://docs.confluent.io/platform/current/schema-registry/avro.html

Bonnes pratiques

1. Schema registry
   - Force la gestion de la compatibilité
2. 1 topic pour 1 ressource ; 1 message = 1 opération sur la ressource (cf ReST)
   - Ex : Create Abonnement
3. Clé = Id de ressource
   - Traitement séquentiel de tous les messages sur une ressource spécifique
   - UUID
4. Petits messages
   - Casser les grappes d’objets, faire un topic par ressource (cf. 2)
5. Concevoir en asynchrone
   - Pouvoir mettre la charrue avant les boeufs
6. Topic de Retry + Dead Letter Queue
   - Monitoring + Alerting

Solutions

Producer : [redacted]

Consumer : [redacted]

Partition & Consumer group :

[redacted]

[redacted]

[redacted]

Schema registry :

[redacted]

[redacted]

[redacted]

[redacted]