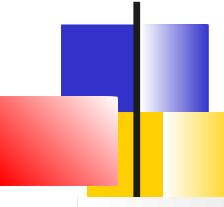


# Spring / Kafka

---

David THIBAU - 2025

david.thibau@gmail.com



# Agenda

## Présentation Kafka

- Le projet Kafka
- Concepts cœur de Kafka, Architecture
- Les différents cas d'usage de Kafka

## Cluster Kafka

- Nœuds du cluster, Kraft
- Distribution / installation
- Utilitaires Kafka,
- Outils graphiques d'administration

## Apache Kafka et ses APIs

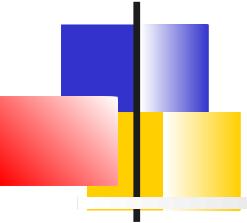
- Producer API
- Consumer API
- Schema Registry et Avro
- Autres APIS, KafkaAdmin et KafkaStream

## Spring Kafka

- Introduction
- Production de messages
- Consommation de messages
- Transaction et sémantique Exactly Once
- Sérialisation / Désérialisation
- Traitement des Exceptions

## Compléments

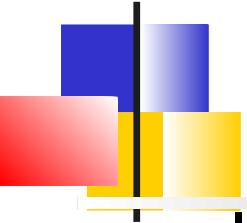
- Tests
- Configuration des listeners
- Mise en place SSL/TLS
- Authentification via SASL
- ACLs
- Quotas



# Présentation Kafka

---

**Le projet Kafka**  
Cas d'usage  
Concepts cœur de Kafka



# Origine

---

Initié par *LinkedIn*, mis en OpenSource en 2011

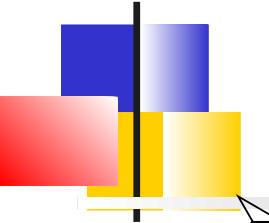
Écrit en *Scala* et *Java*

Au départ, un message broker gérant une file de messages

A évolué pour devenir une plate-forme de streaming d'événements temps-réel

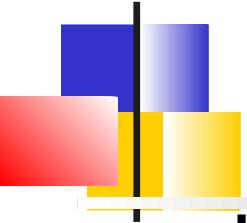
Organisé en cluster, taillé pour le BigData, il est basé sur l'abstraction « d'un journal de commit distribué »

Maintenu par *Confluent* depuis 2014



# Objectifs

- Découpler producteurs et consommateurs de messages
- Persister les messages afin qu'ils puissent être consommés par de nombreux consommateurs, (éventuellement à posteriori)
- Atteindre de très haut débit et une latence faible
- Scaling horizontal flexible
- Offrir des garanties de fiabilité de la livraison de messages, malgré des défaillances !



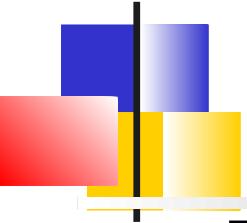
# Fonctionnalités

---

Kafka a trois capacités clés:

- Publier et s'abonner à des flux de messages<sup>1</sup> avec certaines garanties de fiabilité.
- Stocker les flux de messages de manière durable et tolérante aux pannes.
- Traiter, transformer les flux de messages au fur et à mesure qu'ils se produisent.

1. Dans la suite des slides on utilise de façon non-différenciés les termes *message, événement, enregistrement*



# Points forts

---

Très bonne scalabilité et flexibilité

- Gestion des abonnements multiples
- Facilité d'extension du cluster

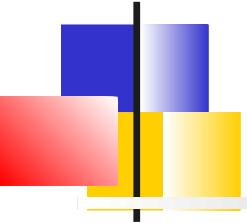
Très bonne performance

Disponibilité et tolérance aux fautes

Rétention sur disque

Traitements distribués d'événements

Intégration avec les autres systèmes



# Confluent

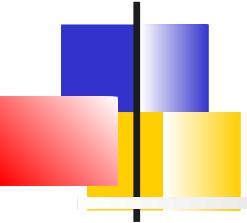
---

Créé en 2014 par *Jay Kreps, Neha Narkhede, et Jun Rao*

Mainteneur principal d'Apache Kafka

Plate-forme *Confluent* :

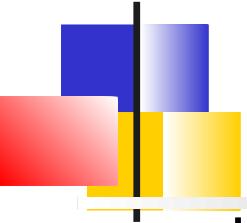
- Une distribution de Kafka
- Fonctionnalités commerciales additionnelles
- Services Cloud



# Présentation Kafka

---

Le projet Kafka  
**Cas d'usage**  
Concepts cœur de Kafka



# Kafka vs Message broker traditionnel

---

Kafka peut être utilisé comme **message broker** permettant de découpler un service producteur de services consommateurs

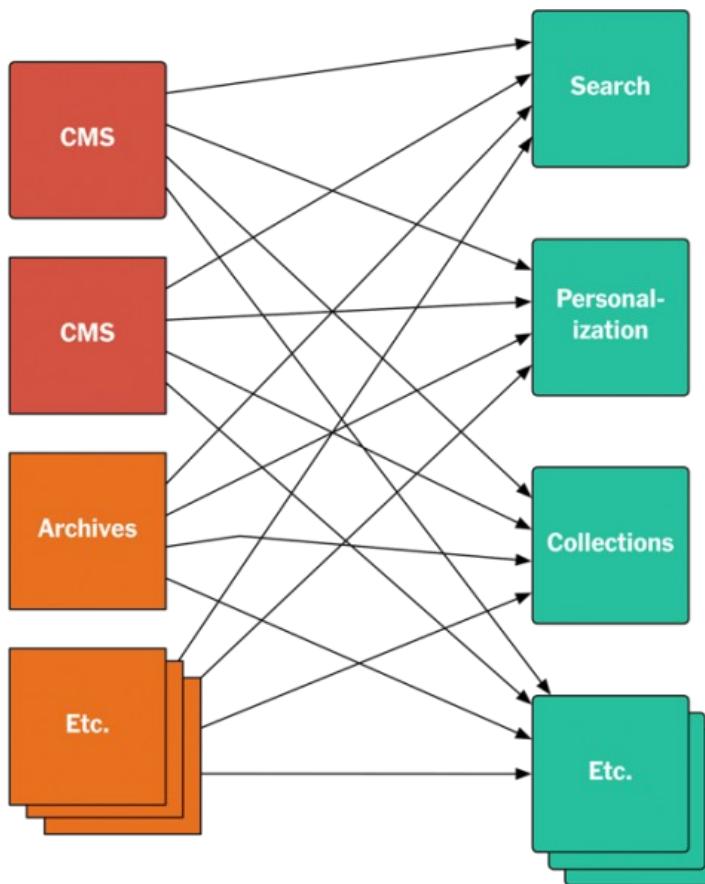
- Kafka n'offre que le modèle **PubSub**.
- Grâce au concept de **groupe de consommateurs**, ce modèle est scalable
- Kafka offre une **garantie plus forte** sur la livraison des messages malgré les défaillances
- Kafka ne supprime pas les messages après consommation. Ils peuvent être **consommés à posteriori**

=> Idéal comme plate-forme d'intégration entre services :  
Architecture micro-services, ESB, EAI

# Exemple ESB (New York Times)

## Avant

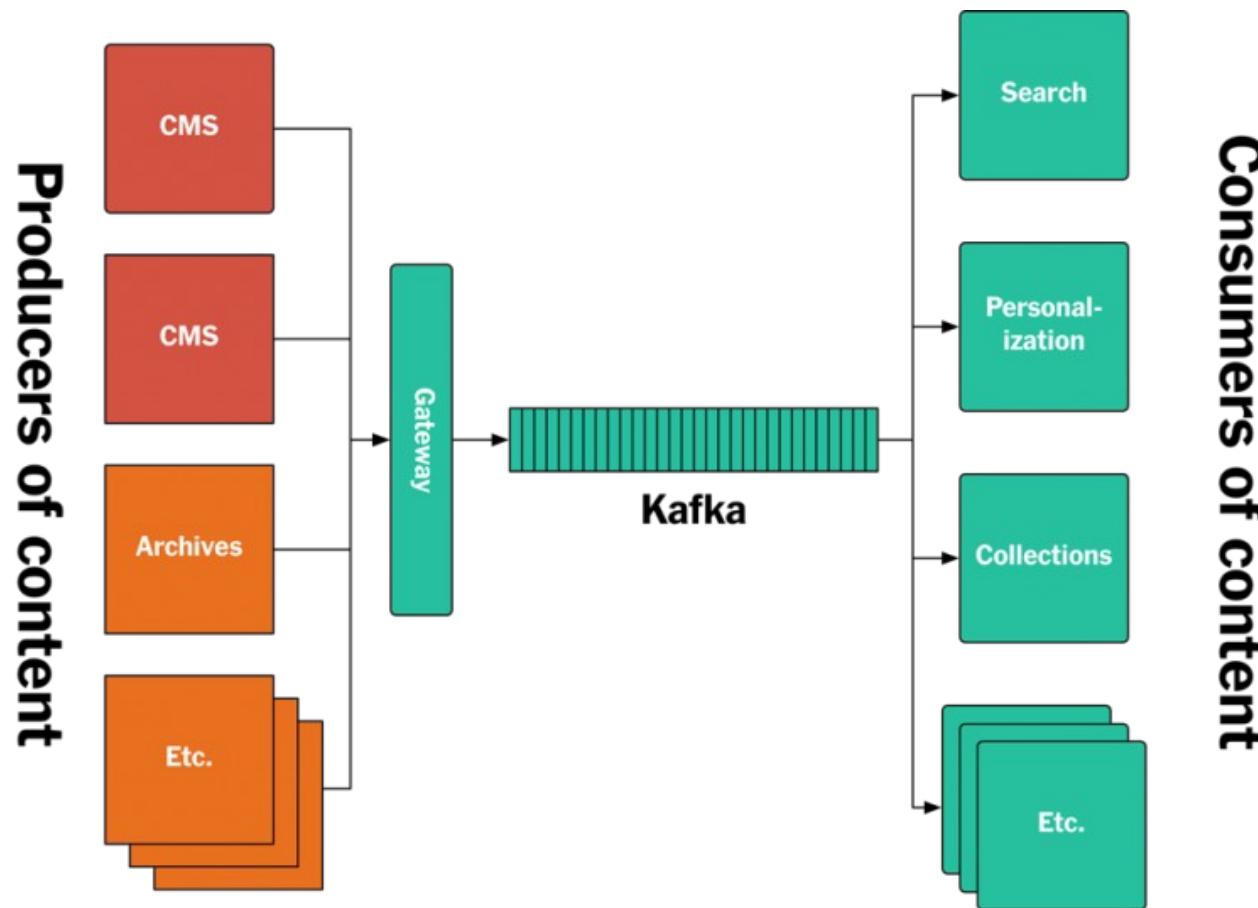
Producers of content



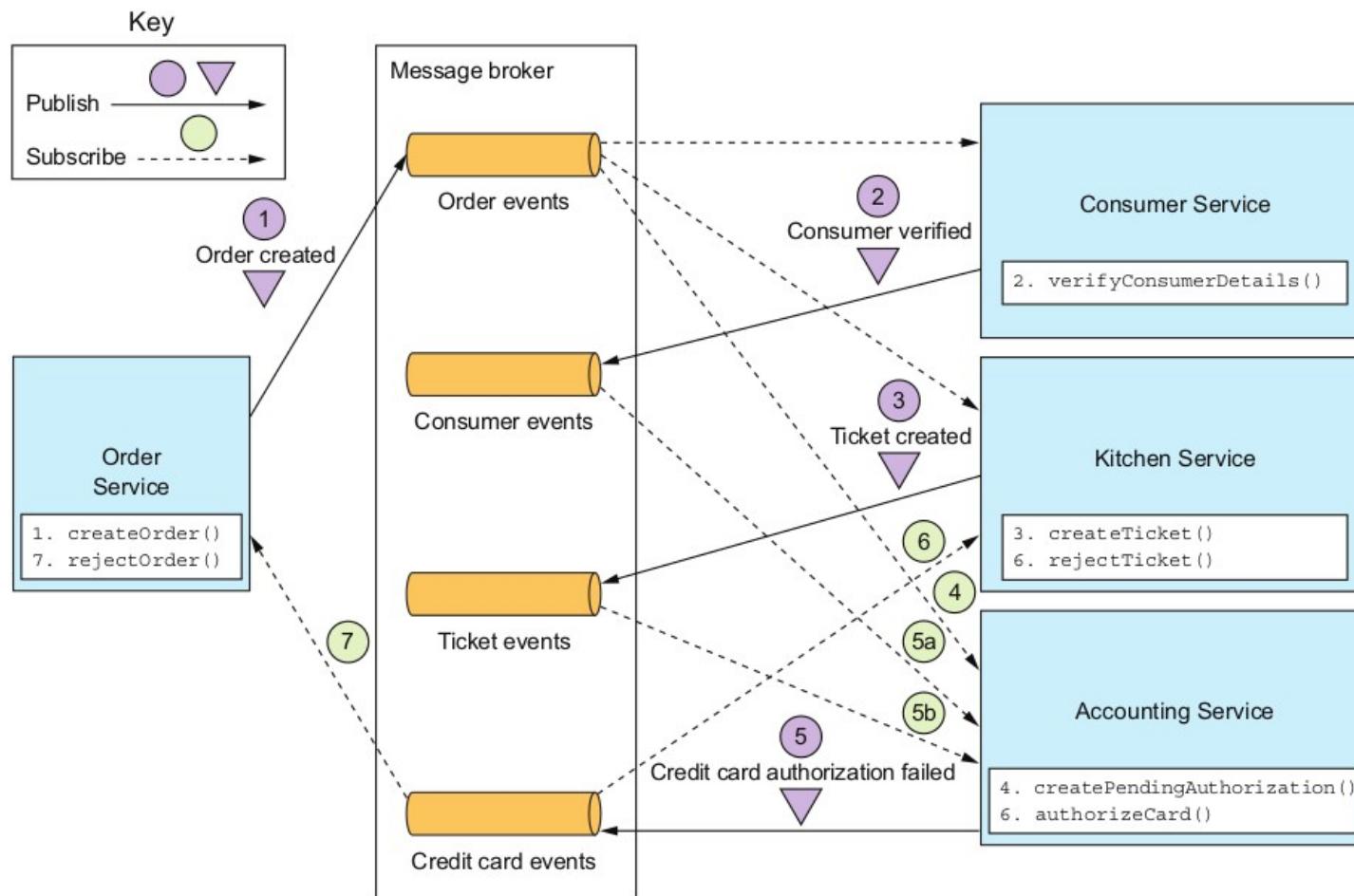
Consumers of content

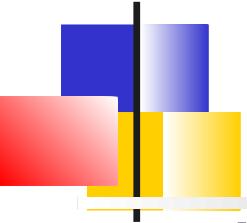
# Exemple ESB (New York Times)

## Après



# Exemple Micro-services SAGA pattern





# Kafka pour l'ingestion massive de données

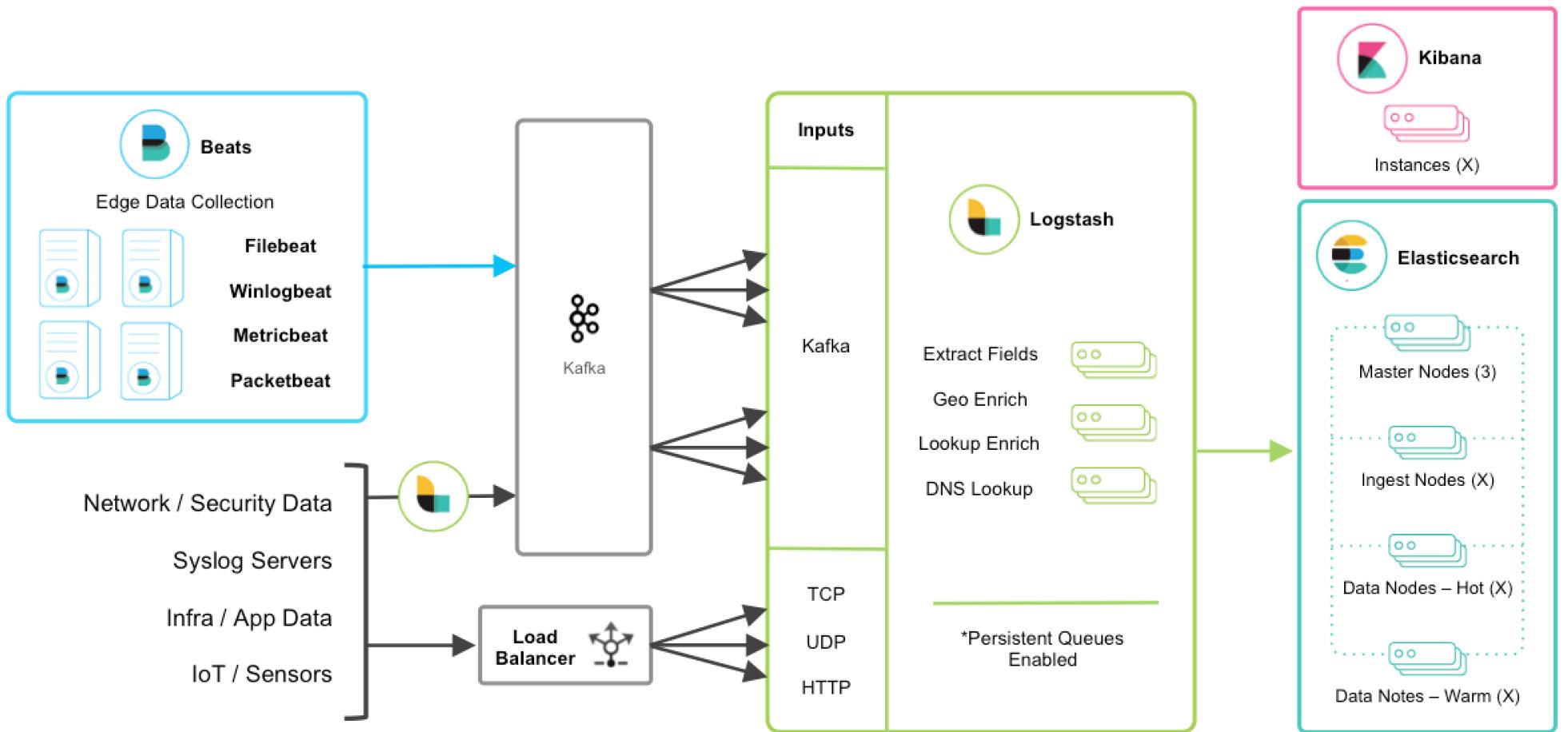
---

Les *topics* Kafka peuvent être utilisés pour bufferiser les événements provenant des producteurs afin que les consommateurs puissent supporter des débits de production élevé

Typique des architectures BigData, d'analyse ou du monitoring temps réel.

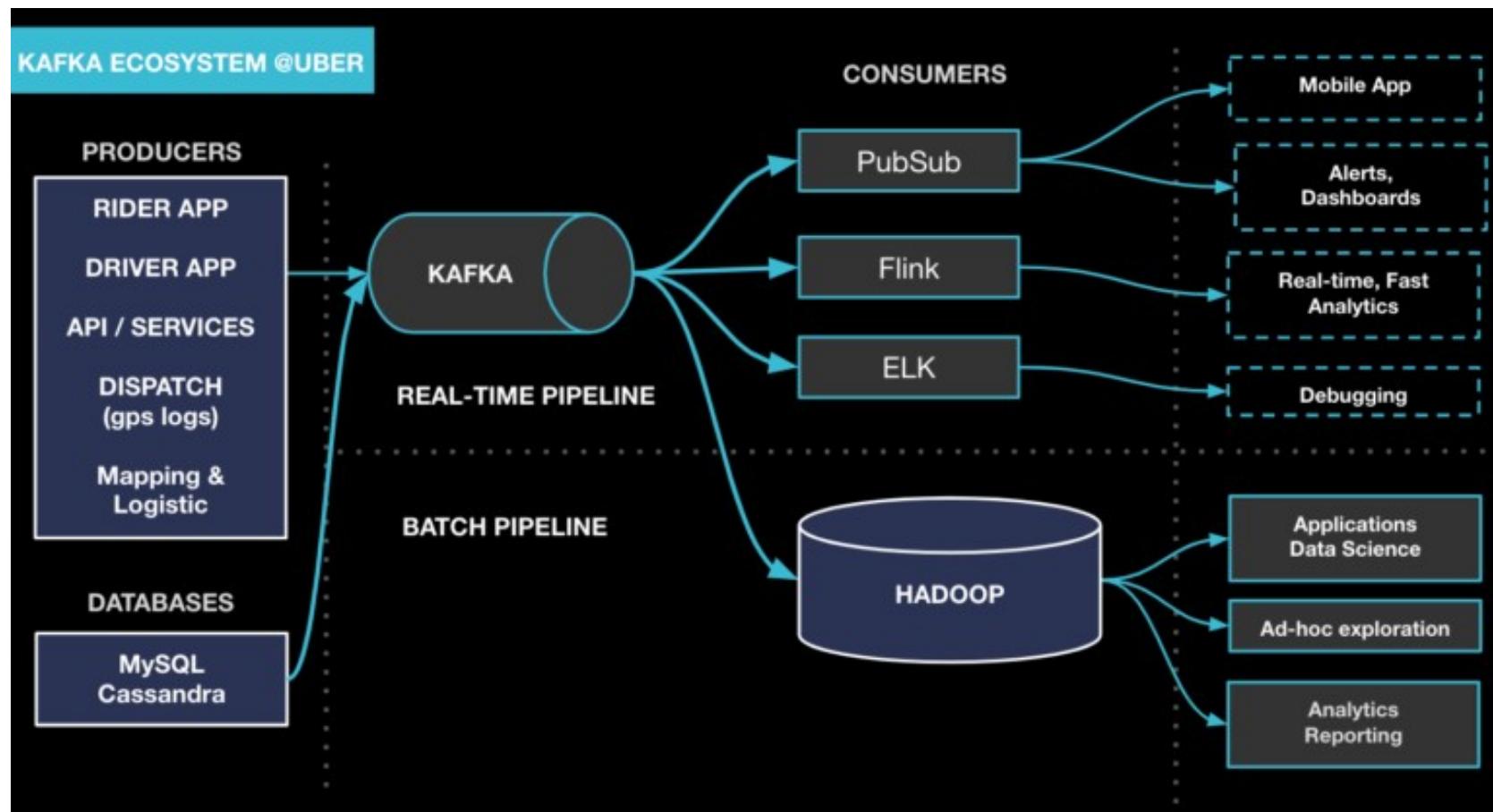
# Exemple Architecture ELK

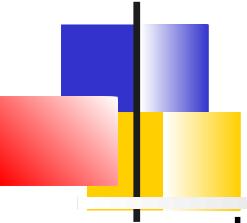
## Bufferisation des événements



# Ingestion massive de données

## Exemple Uber





# Architecture Event-driven

---

Les architectures *event-driven* sont une alternative intéressante dans un contexte micro-services.

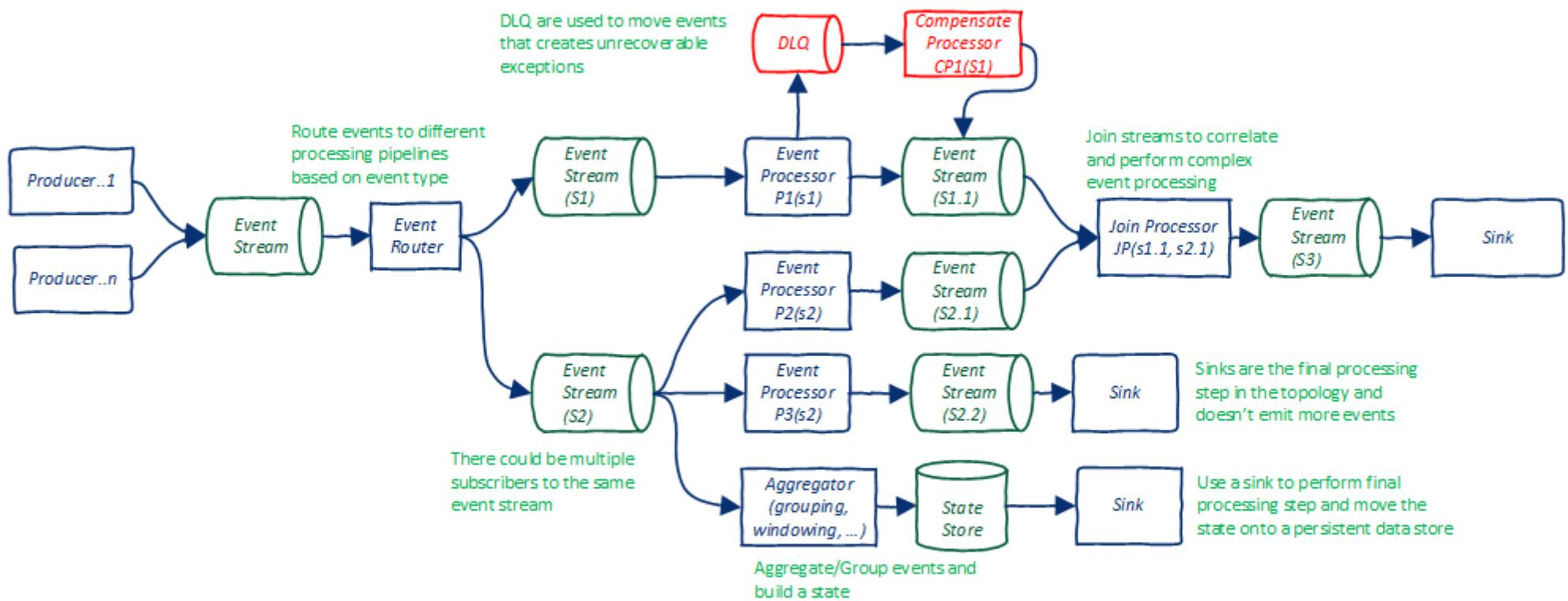
- Cela produit généralement des architectures plus souples et plus réactives.

Chaque micro-service consomme en continu des événements :

- Lit un ou plusieurs topics Kafka en entrée
- Effectue un traitement
- Écrit vers un ou plusieurs topics de sortie

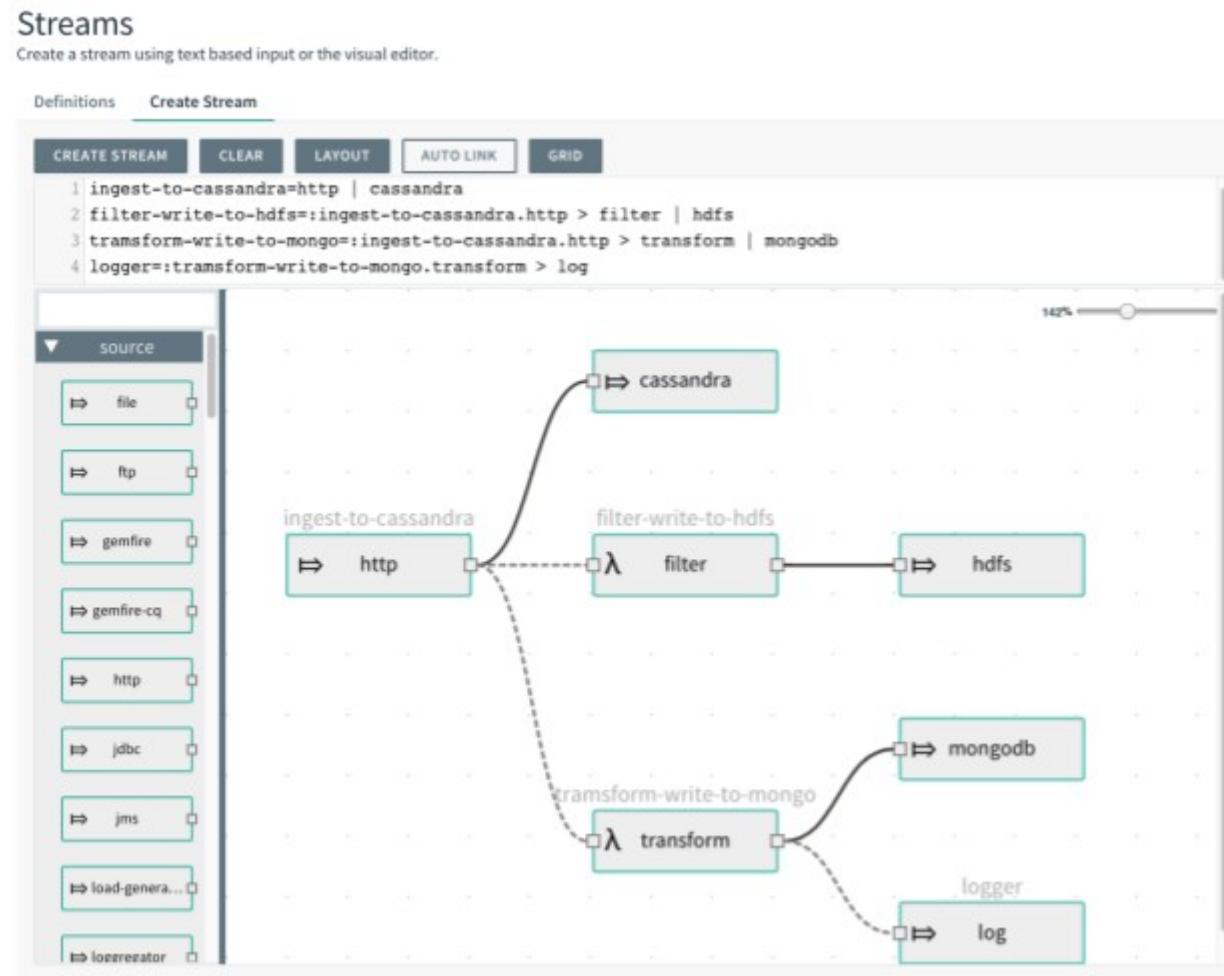
*Kafka Stream, Spring Cloud Stream ou Spring Cloud Data Flow* facilitent ce type d'architecture

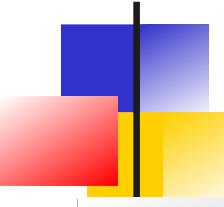
# Event-driven architecture



Event Processor – could be a microservice, serverless function, etc – which implements the logic of a particular processing step

# Data Stream Exemple Spring Cloud Data Flow

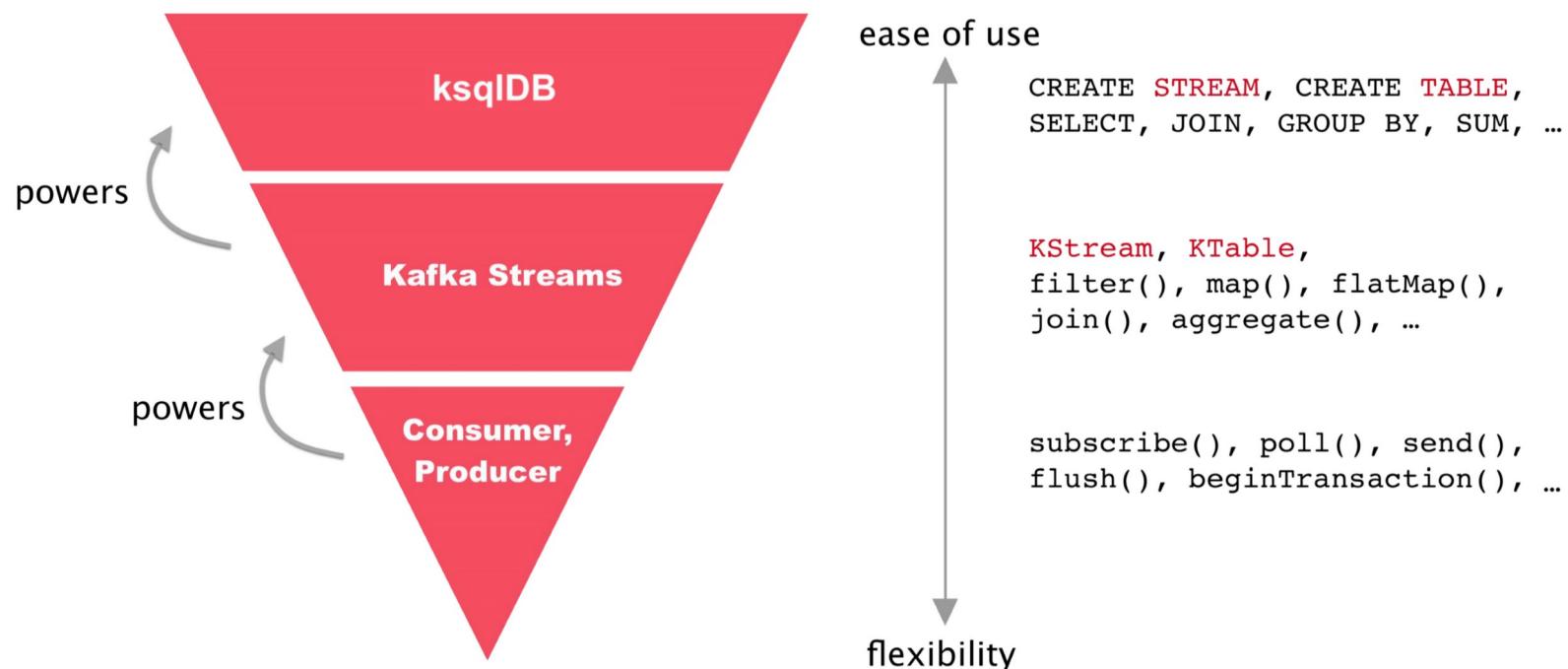




# ksqldb

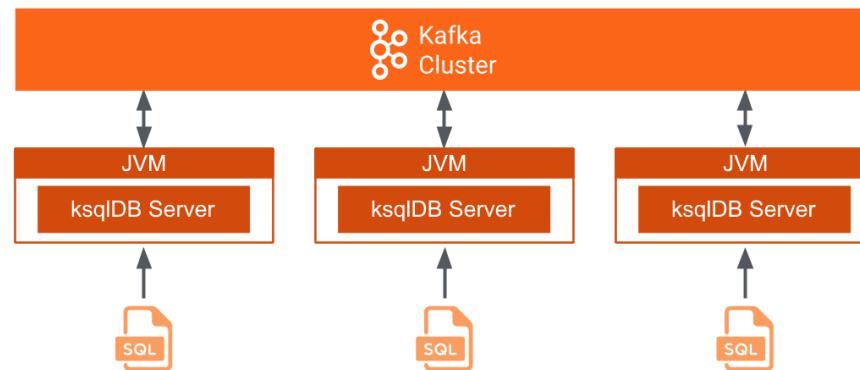
---

*ksqldb* est une abstraction au dessus de KafkaStream permettant de bâtir ces applications via des instructions SQL



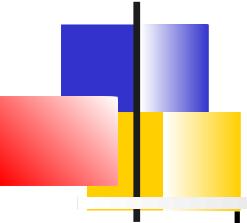
# Architecture ksqlDB

ksqlDB Standalone Application (Headless Mode)



## Intégration via

- API REST et ksqlCli
- Librairies : Clients Java, Python, NodeJS



# Kafka comme système de stockage

---

Les enregistrements sont écrits et répliqués sur le disque.

La structure de stockage est très scalable.

Kafka fonctionne de la même manière avec 50 Ko ou 50 To de données.

=> Kafka peut être considéré comme un système de stockage.

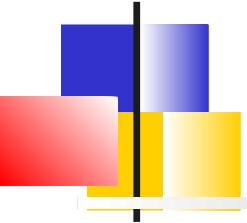
A la différence d'une BD, il stocke l'intégralité de l'historique des données plutôt qu'un simple instantané

Kafka peut alors être utilisé comme *EventStore* et permet la mise en place du pattern *Event Sourcing*<sup>1</sup> utilisé dans les micro-services

Des abstractions sont proposées pour faciliter la manipulation de l'*EventStore* : Projet **ksqldb**<sup>2</sup>

1. <https://microservices.io/patterns/data/event-sourcing.html>

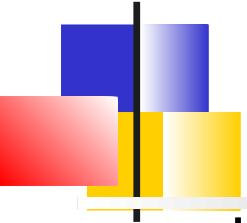
2. <https://ksqldb.io/overview.html>



# Présentation Kafka

---

Le projet Kafka  
Cas d'usage  
**Concepts cœur de Kafka**



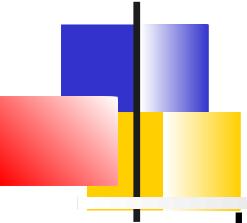
# Concepts de base

---

Kafka s'exécute en **cluster** sur un ou plusieurs serveurs (**brokers**) pouvant être déployés dans différents data-center.

Le cluster Kafka stocke des flux d'enregistrements : les **records** dans des rubriques : les **topics** .

Chaque enregistrement se compose d'une clé éventuelle, d'une valeur, d'un horodatage et éventuellement des entêtes.



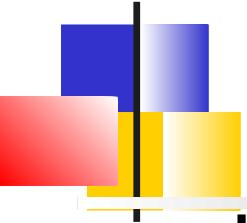
# Cluster

---

Kafka est exécuté comme un cluster d'un ou plusieurs **serveurs** pouvant s'étendre sur plusieurs centres de données.

- Certains de ces serveurs appelés les **brokers** forment la couche de stockage.
- Un serveur est désigné **contrôleur**. Son rôle est de prendre des décisions concernant le cluster comme l'affectation de partitions

Avant la version 3.x, un cluster Kafka nécessitait également un ensemble Zookeeper faisant office de contrôleur



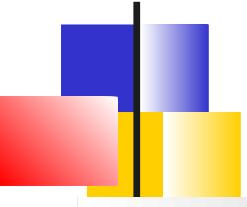
# Configuration des noeuds

---

Les configurations principales sont :

- ***cluster.id***<sup>1</sup> : Identique pour chaque broker.
- ***broker.id/node.id*** : Différent pour chaque broker
- ***log.dirs*** : Ensemble de répertoires de stockage des enregistrements
- ***listeners*** : Ports ouverts pour communication avec les clients et inter-broker

1. Généré automatiquement via Zookeeper, doit être précisé en mode Kraft



# Protocole Client/Serveur

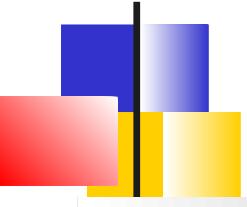
---

Dans Kafka, la communication entre les clients et les serveurs s'effectue via un protocole TCP simple, performant et indépendant du langage.

- Ce protocole est versionné et maintient une compatibilité ascendante avec les versions plus anciennes.

Apache fournit un client Java, mais les clients sont disponibles dans de nombreuses langages.<sup>1</sup>

1. <https://cwiki.apache.org/confluence/display/KAFKA/Clients>



# Topic

---

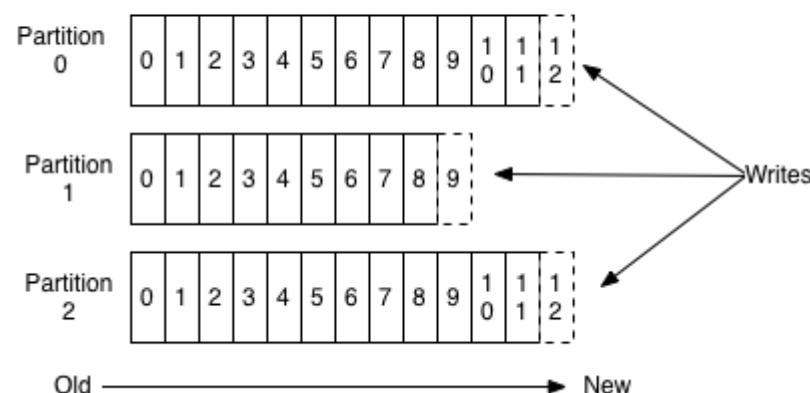
Les *records* sont publiés vers des **topics**.

Les *topics* de Kafka peuvent avoir Zéro, Un ou de multiples abonnés

Les topics peuvent être **partitionnés**.

Le cluster Kafka conserve donc un journal partitionné

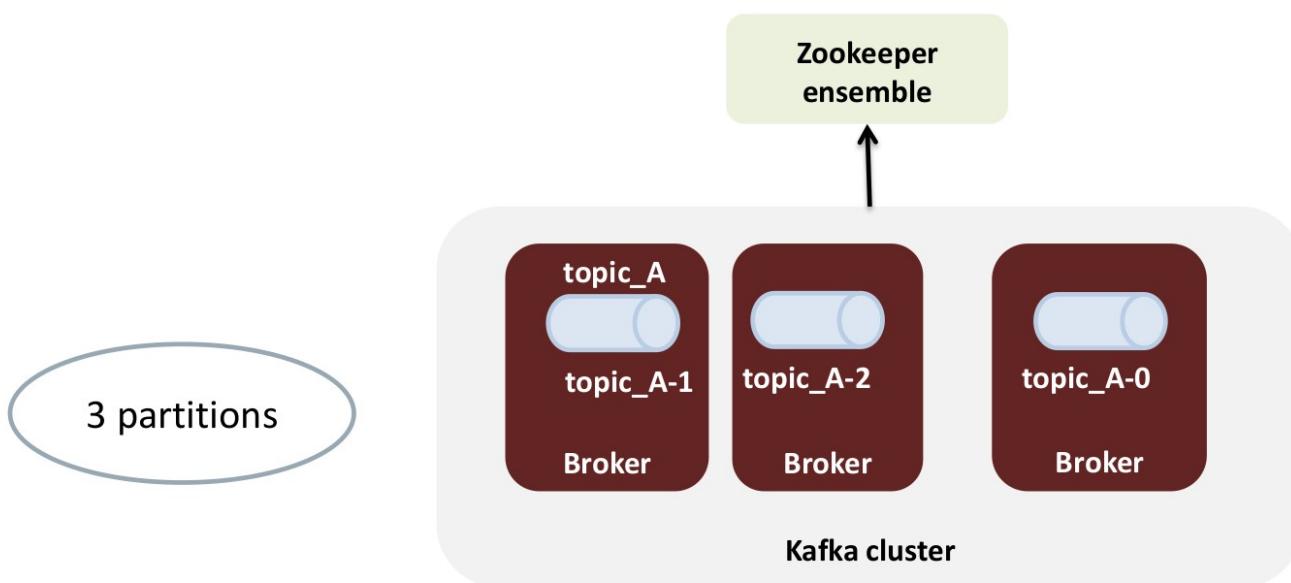
Anatomy of a Topic



# Apport des partitions

Les partitions autorisent le **parallélisme** et augmentent la **capacité de stockage** en utilisant les capacités disque de plusieurs brokers.

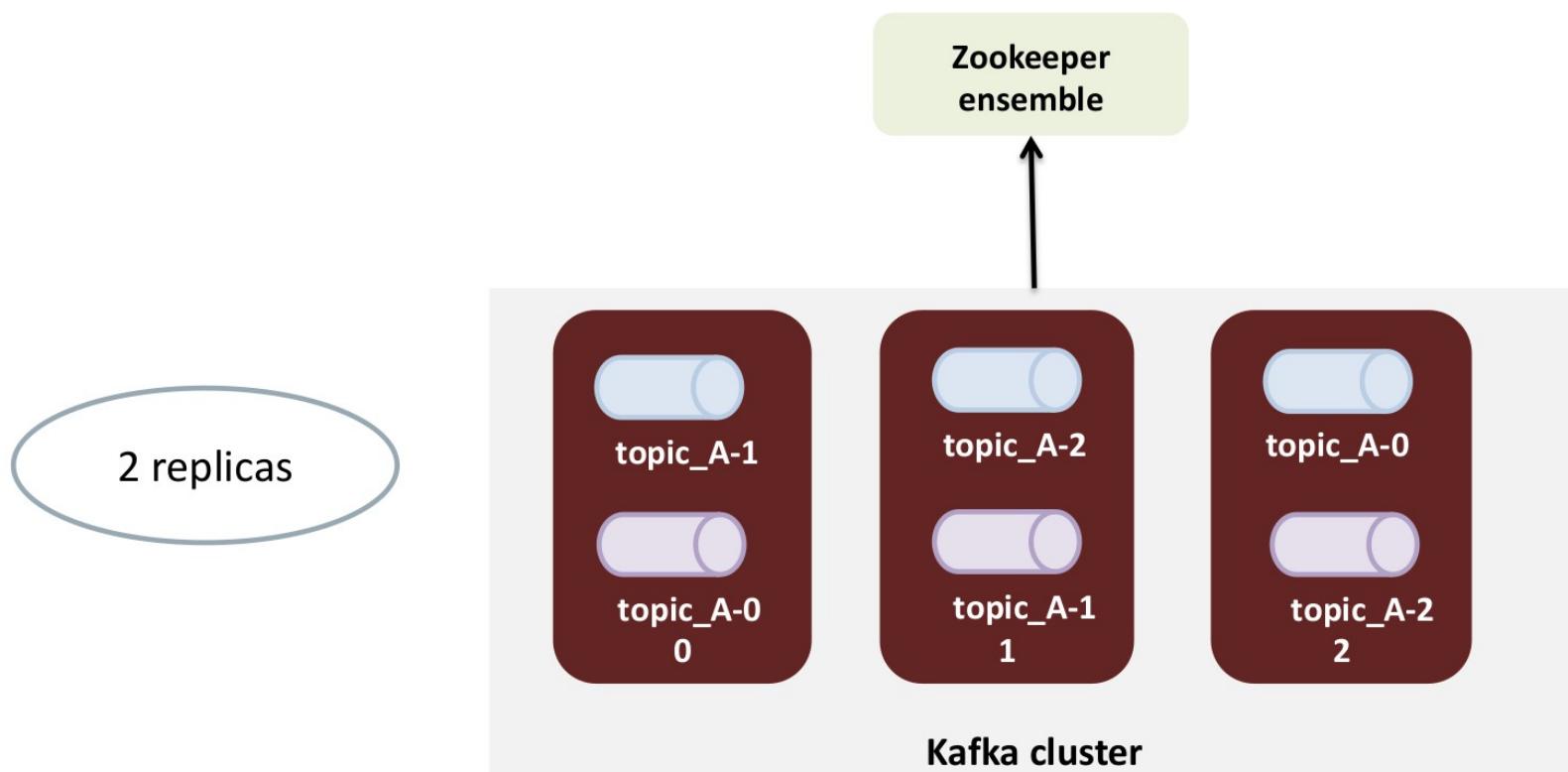
L'ordre des messages n'est garanti qu'à l'intérieur d'une partition  
Le nombre de partition est fixé à la création du *topic*

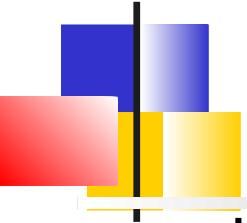


# RéPLICATION

Les partitions peuvent être **répliquées**

- La réPLICATION permet la tolérance aux pannes et la durabilité des données





# Distribution des partitions

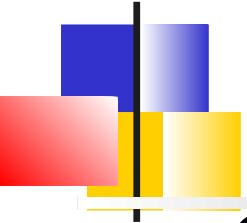
---

Les partitions sont réparties sur les instances du cluster.

Les répliques sont distribuées sur des instances différentes

Pour chaque partition répliquée, une des instances agit comme **maître (leader)**. Les autres comme **suiveurs (follower)**

- Le maître coordonne les lectures et les écritures sur la partition
- Les suiveurs répliquent passivement le maître
- Si le maître défaillie, un processus d'élection choisit un autre maître parmi les répliques



# Partition et offset

---

Chaque partition est une séquence  
**ordonnée et immuable**  
d'enregistrements.

Un numéro d'identification séquentiel  
nommé **offset** est attribué à chaque  
enregistrement.

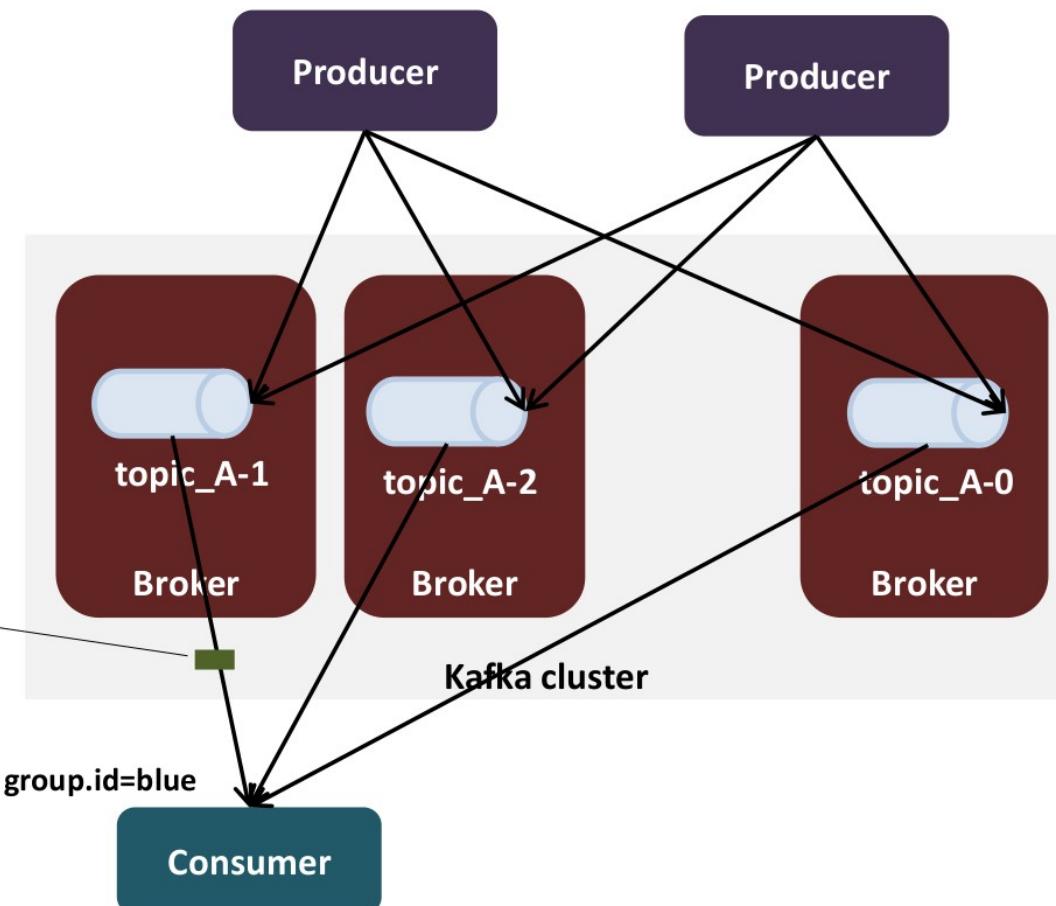
Le cluster Kafka conserve durablement tous  
les enregistrements publiés, qu'ils aient  
ou non été consommés, en utilisant une  
**période de rétention** configurable.

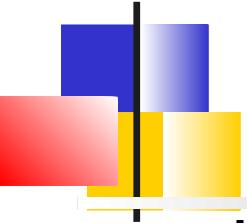
# Clients du cluster

Les producteurs et consommateurs sont connectés à tous les brokers détenant le *topic*

Ils découvrent les nœuds grâce à des adresses de *bootstrap*

- message (record, event)
  - key-value pair
  - string, binary, json, avro
  - serialized by the producer
  - stored in broker as byte arrays
  - deserialized by the consumer





# Routing des messages

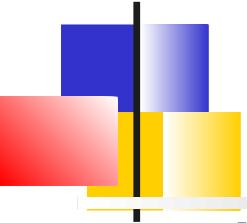
---

Les producteurs sont responsables du choix de la partition en fonction de l'enregistrement

Cela peut être fait

- Via une stratégie Round-Robin assurant un équilibrage de charge
- En fonction des données de l'enregistrement. Typiquement, la clé

Ce routage est en général délégué au client Kafka mais peut également être pris en charge par l'application.

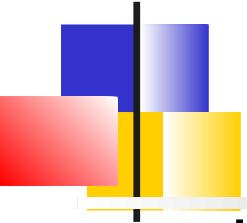


# Groupe de consommateurs

---

Les consommateurs sont taggés avec un nom de **groupe**

- Plusieurs instances de processus ou plusieurs threads peuvent avoir le même nom de groupe  
=> scalabilité de la consommation
- Chaque enregistrement d'un topic est remis à une instance au sein de chaque groupe de consommateurs.



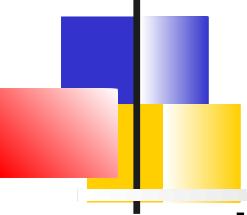
# Offset consommateur

---

Kafka conserve pour un groupe de consommateurs son **offset** de lecture.

Kafka met à jour l'offset lorsqu'il reçoit des ordres de **commit** de la part du consommateur:

- normalement, le consommateur avance son offset au fur et à mesure de sa lecture des enregistrements,
- mais, il peut consommer dans l'ordre qu'il souhaite.  
Par exemple, retraitier les données les plus anciennes ou repartir d'un offset particulier.



# Consommateur vs Partition

## Rééquilibrage dynamique

---

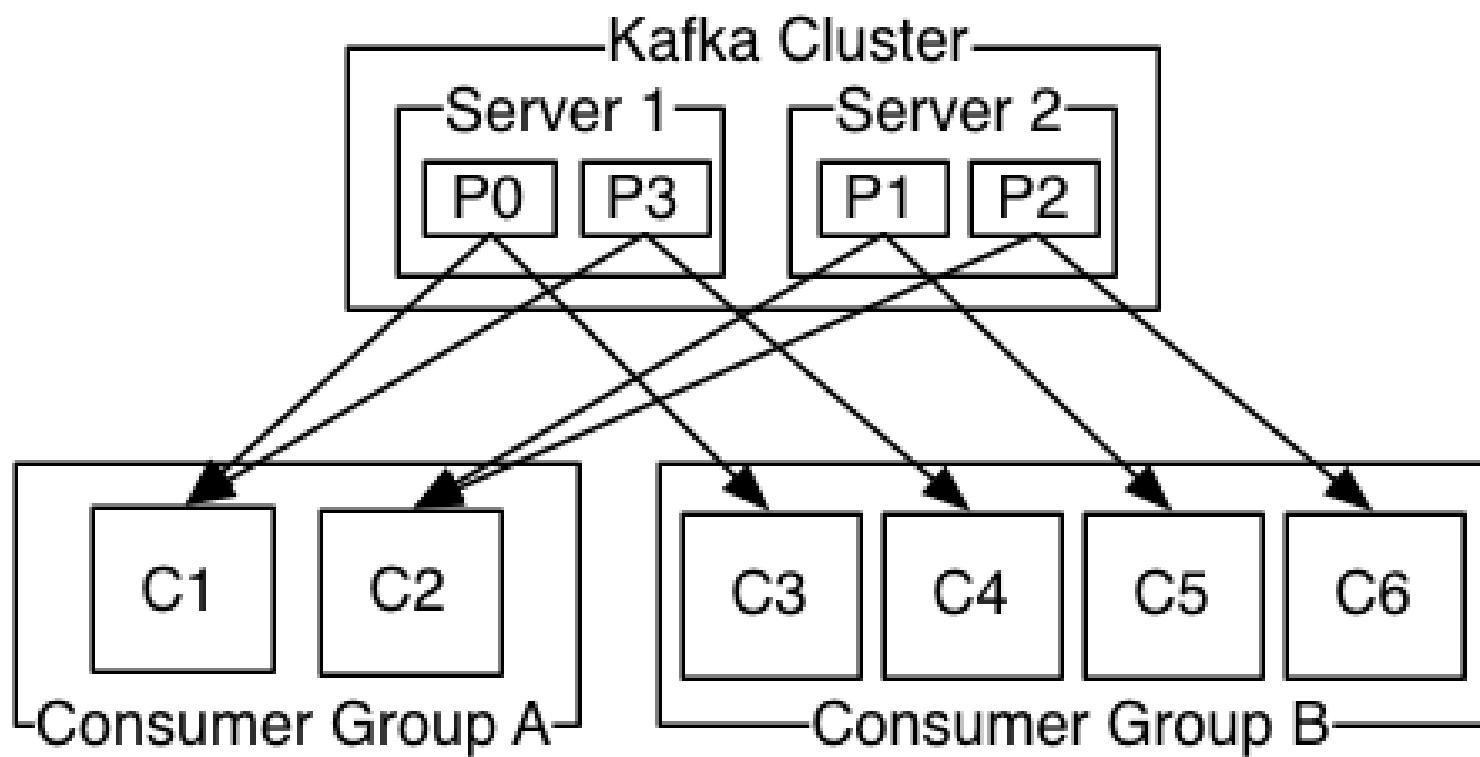
Kafka assigne les partitions à des instances de consommateur d'un même groupe.

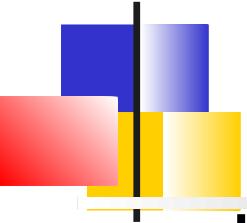
- A tout moment, un consommateur est exclusivement dédié à une partition

Kafka gère des rééquilibrages.

- Si de nouvelles instances rejoignent le groupe, elles reprendront certaines partitions des autres membres du groupe;
- Si une instance meurt, ses partitions seront distribuées aux instances restantes.

# Exemple



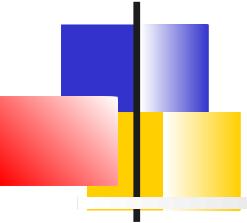


# Ordre des enregistrements

---

Kafka garantit un ordre total sur les enregistrements d'une partition, mais pas sur les différentes partitions d'un topic.

- L'ordre sur les partitions, combiné à la possibilité de partitionner les données par une clé est suffisant pour la plupart des applications.
- Si une application nécessite un ordre strict sur tous les enregistrements. Il faut que le topic n'est qu'une seule partition



# Apache Kafka et ses APIs

---

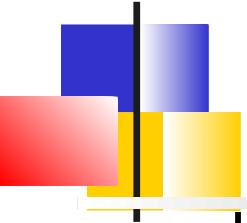
## **Les APIs Kafka**

*Producer API*

*Consumer API*

*Sérialisation et Schema Registry*

*Autres APIs*



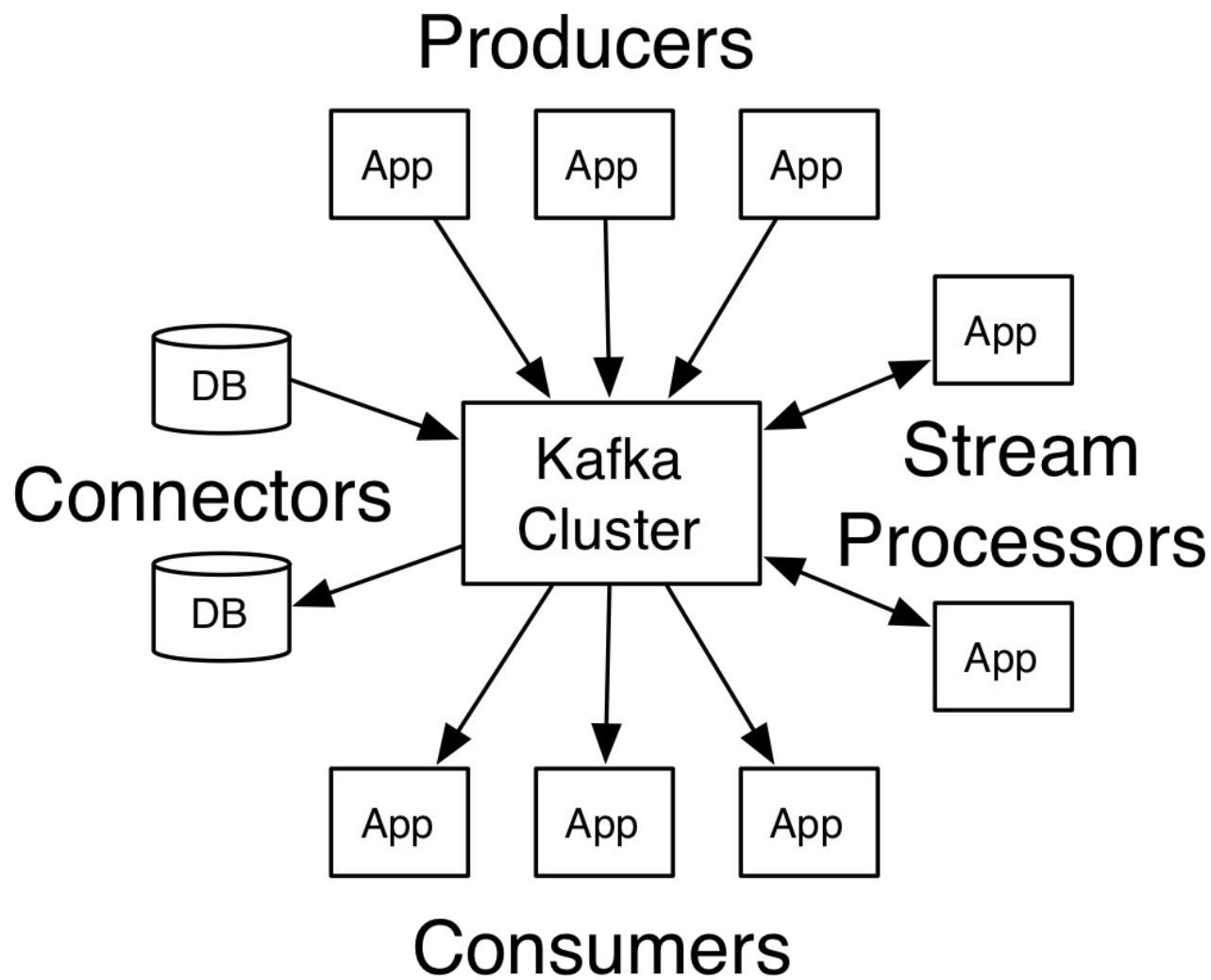
# APIs

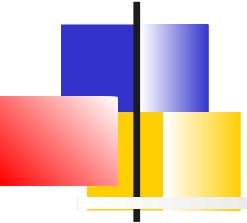
---

Kafka propose 5 principales APIs :

- L'API **Producer** permet à une application de publier un flux sur un ou plusieurs topics Kafka.
- L'API **Consumer** permet à une application de s'abonner à un ou plusieurs topics et de traiter le flux d'enregistrements associé.
- L'API **Streams** permet à une application d'agir comme un processeur de flux, consommant un ou plusieurs topic d'entrée et produisant un flux de sortie vers un ou plusieurs topics.
- L'API **Connector** permet de créer et d'exécuter des producteurs ou des consommateurs à partir de système tierces (BD, fichiers, STDOUT, ...)
- L'API **Admin** permet de gérer les topics et le cluster

# APIs

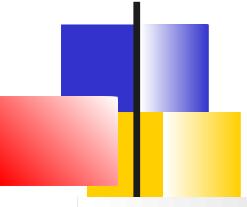




# Apache Kafka et ses APIs

---

Les APIs Kafka  
**Producer API**  
Consumer API  
Sérialisation Json, Avro  
KafkaAdmin et KafkaStream



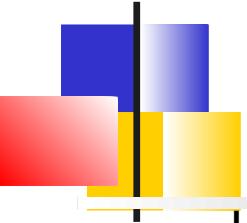
# Introduction

---

L'API est simple mais les contraintes applicatives influencent la façon de l'utiliser ainsi que la configuration des *topics*

Les questions devant être posées :

- Chaque message est-il critique, ou peut-on tolérer des pertes de messages?
- La duplication accidentelle de messages est-elle autorisée ?
- Y-a-t-il des exigences strictes de latence ou de débit ?



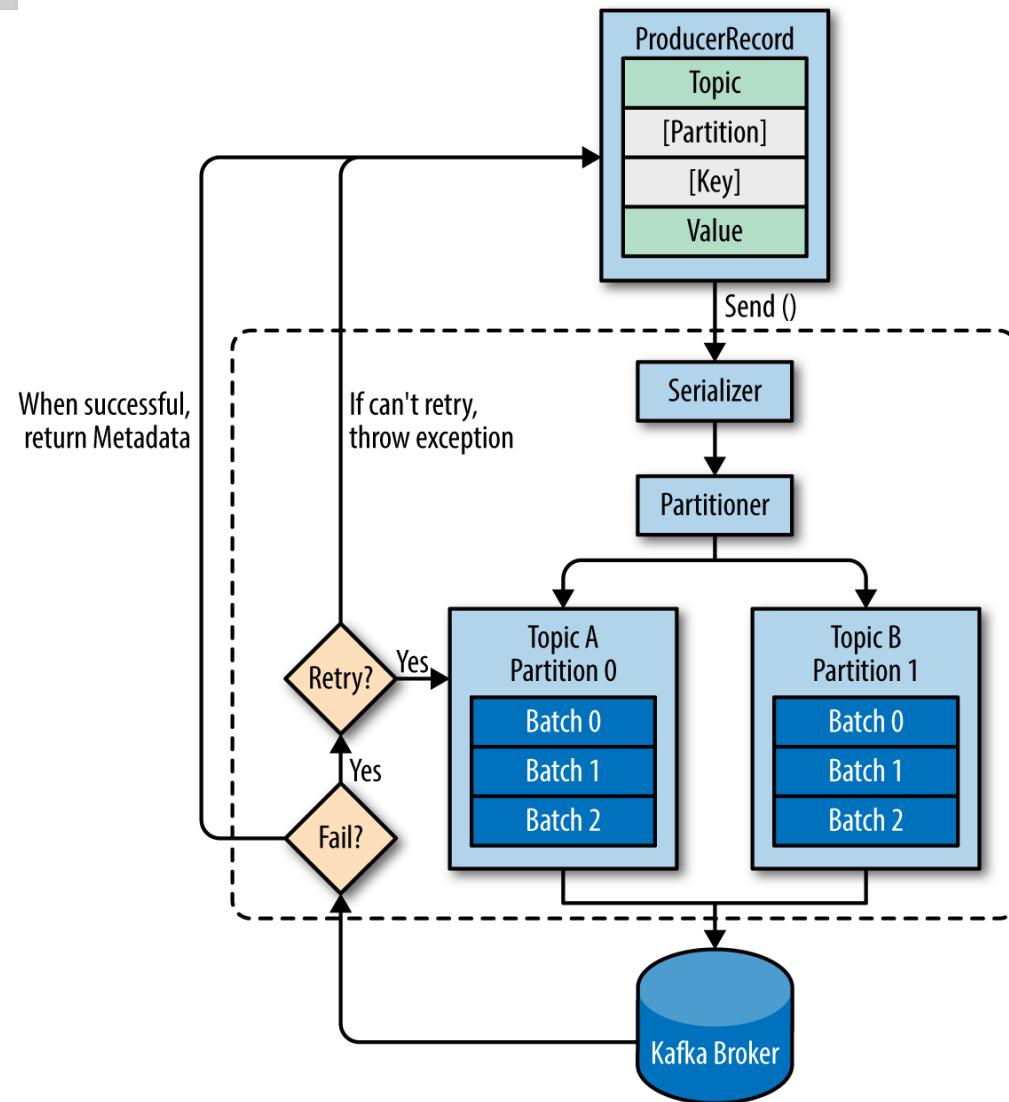
# Étapes lors de l'envoi d'un message

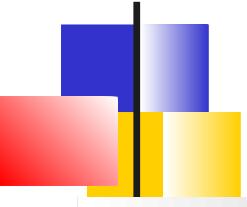
---

L'envoi de message est constitué de plusieurs étapes :

- Création d'un objet **ProductRecord** encapsulant le topic et optionnellement une clé et une partition
- L'objet est **sérialisé** pour préparer sa transmission sur le réseau
- Les données sont ensuite fournies à un **partitionneur** qui détermine la partition de destination, (à partir de la partition indiquée, de la clé du message ou en Round-robin)
- Une fois la partition sélectionnée, le message est ajouté à un **lot de messages** destiné à la même partition.  
Une thread séparée envoie le lot de messages.
- Lorsque le broker reçoit le message, il renvoie une réponse sous la forme d'un objet **RecordMetadata** encapsulant le *topic*, la partition, la clé et l'offset
- Si le broker n'arrive pas à écrire le message, il renvoie une erreur et le producteur peut réessayer un certain nombre de fois si l'erreur est **Retriable**

# Envoi de message





# Construire un Producteur

---

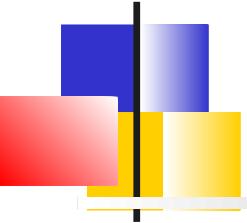
La première étape consiste à instancier un **KafkaProducer** en lui passant des propriétés de configuration

3 propriétés de configurations sont obligatoires :

- **bootstrap.servers** : Liste de brokers que le producteur contacte pour découvrir le cluster
- **key.serializer** : La classe utilisée pour la sérialisation de la clé
- **value.serializer** : La classe utilisée pour la sérialisation du message ...

1 optionnelle est généralement positionnée :

- **client.id** : Permet de tracer le producteur de message

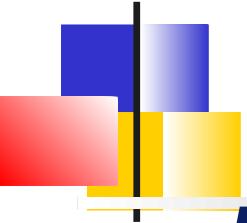


# Exemple Java

---

```
private Properties kafkaProps = new Properties();
kafkaProps.put("bootstrap.servers",
    "broker1:9092,broker2:9092");
kafkaProps.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");
kafkaProps.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");

producer = new KafkaProducer<String, String>(kafkaProps);
```



# *ProducerRecord*

**ProducerRecord** représente l'enregistrement à envoyer .

Il contient le nom du *topic*, une valeur et éventuellement une clé, une partition, un timestamp

Constructeurs disponibles :

# Sans clé

```
ProducerRecord(String topic, V value)
```

# Avec clé

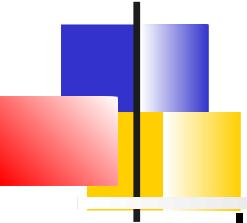
```
ProducerRecord(String topic, K key, V value)
```

# Avec clé et partition

```
ProducerRecord(String topic, Integer partition, K key, V value)
```

# Avec clé, partition et timestamp

```
ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value)
```

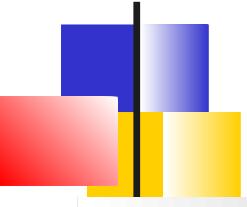


# Méthodes d'envoi des messages

---

Il y a 3 façons d'envoyer des messages :

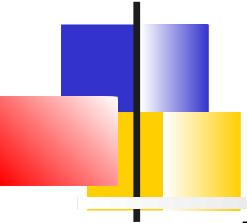
- **Fire-and-forget** : On n'attend pas d'acquittement,
- **Envoi synchrone** : La méthode d'envoi retourne un *Future*, l'appel à *get()* est bloquant et contient la réponse du broker. On traite éventuellement les cas d'erreurs
- **Envoi asynchrone** : Lors de l'envoi, on passe en argument une fonction de callback.  
La méthode est appelée lorsque la réponse est retournée



# Méthodes d'envoi

---

```
ProducerRecord<String, String> record =  
new ProducerRecord<>("CustomerCountry", "Precision", "France");  
// Fire and forget  
try {  
    producer.send(record);  
} catch (Exception e) { e.printStackTrace(); }  
  
// Envoi synchrone  
try {  
    producer.send(record).get();  
} catch (Exception e) {e.printStackTrace(); }  
  
// Asynchrone avec Callback  
producer.send(record, new Callback(){  
    @Override  
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {  
        ...  
    }  
});
```

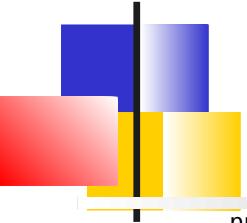


# Sérialiseurs

---

Kafka inclut des sérialiseurs pour les types basiques (***ByteArraySerializer***, ***StringSerializer***, ***LongSerializer***, etc.).

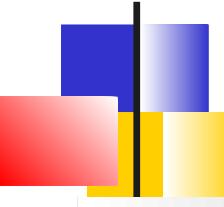
Pour des objets du domaine, il faut implémenter ses propres sérialiseurs/désérialiseurs en s'appuyant sur des librairies comme *Avro*, *Thrift*, *Protobuf* ou *Jackson*



# Exemple sérialiseur s'appuyant sur Jackson

---

```
public class JsonPOJOserializer<T> implements Serializer<T> {  
    private final ObjectMapper objectMapper = new ObjectMapper();  
    /**  
     * Default constructor requis par Kafka  
     */  
    public JsonPOJOserializer() { }  
  
    @Override  
    public void configure(Map<String, ?> props, boolean isKey) { }  
  
    @Override  
    public byte[] serialize(String topic, T data) {  
        if (data == null)  
            return null;  
  
        try {  
            return objectMapper.writeValueAsBytes(data);  
        } catch (Exception e) {  
            throw new SerializationException("Error serializing JSON message", e);  
        }  
    }  
  
    @Override  
    public void close() { }  
}
```



# Envoi de message

---

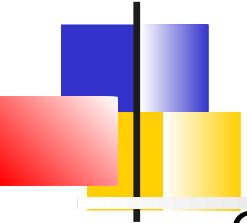
```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("value.serializer",
"org.myappli.JsonPOJOSerializer");

String topic = "customerContacts";

Producer<String, Customer> producer =
    new
KafkaProducer<String,Customer>(props);

Customer customer = CustomerGenerator.getNext();

ProducerRecord<String, Customer> record =
    new ProducerRecord<>(topic, customer.getId(),
customer);
producer.send(record);
```

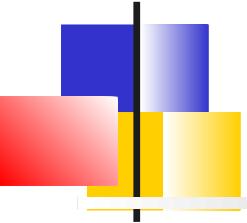


# Configuration des producteurs fiabilité

---

Certains paramètres ont un impact sur la fiabilité des producteurs.

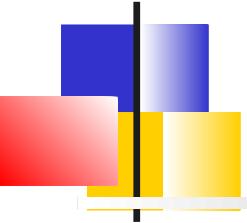
- **acks** : contrôle le nombre de répliques qui doivent recevoir l'enregistrement avant que le producteur puisse considérer l'écriture comme réussie. (All,1 ou 0)
- **retries** : Si l'erreur renvoyée est de type Retriable, le nombre de tentative de renvoi. Si  $> 0$  possibilité d'envoi en doublon ou envoi dans le désordre
- **max.in.flight.requests.per.connection** : Maximum de message en cours de transmission (sans réponse obtenu). Si = 1 garantie d'ordre de l'émission des messages
- **enable.idempotence** : Livraison unique de message, élimine les doublons
- **transactional.id** : Mode transactionnel. Plusieurs envois de messages sont englobés dans une transaction. Permet la sémantique Exactly Once pour les architectures Event-Driven



# Configuration des producteurs *performance*

---

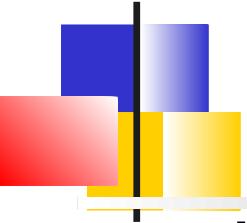
- **batch.size** : La taille du batch en mémoire pour envoyer les messages.  
Défaut 16ko
- **linger.ms** : la durée d'attente de messages supplémentaires avant d'envoyer le batch courant. Défaut 0ms
- **buffer.memory** : Taille buffer pour stocker les messages que l'on ne peut pas envoyé. Défaut 32Mo
- **compression.type** : Par défaut, les messages ne sont pas compressés.  
Valeurs possibles : *snappy* , *gzip* , ou *lz4*
- **request.timeout.ms**, **metadata.fetch.timeout.ms** et **timeout.ms**:  
*Timeouts pour la réception d'une réponse à un message, pour obtenir des méta-données (leader, etc..) pour obtenir le ack des répliques.*
- **max.block.ms** : *Temps maximum d'attente pour la méthode send(). Dans le cas où le buffer est rempli*
- **max.request.size** : *Taille max d'un message*
- **receive.buffer.bytes** et **send.buffer.bytes**: *Taille des buffers TCP*



# Apache Kafka et ses APIs

---

Les APIs Kafka  
Producer API  
**Consumer API**  
Sérialisation et Schema Registry  
Autres APIs

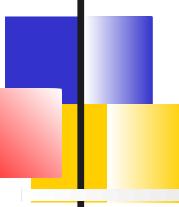


# Introduction

---

Les applications qui ont besoin de lire les données de Kafka utilisent un **KafkaConsumer** pour s'abonner aux topics

Pour bien comprendre l'API, il faut comprendre la notion de groupe de consommateurs et leurs relations avec les partitions

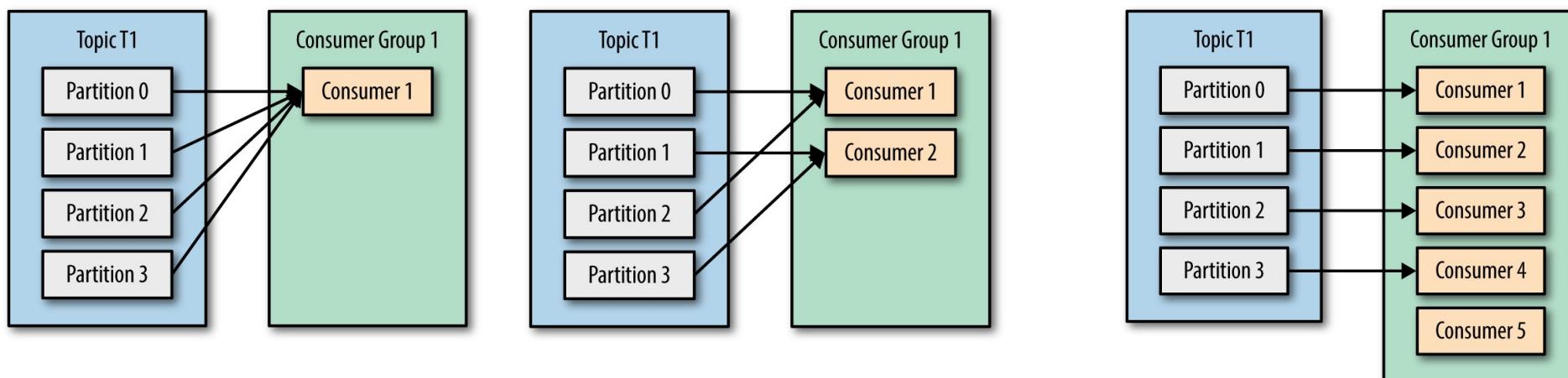


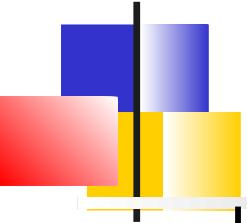
# Groupes de consommateurs

---

Les consommateurs font généralement partie d'un **groupe de consommateurs**.

- Chaque consommateur d'un groupe reçoit les messages d'un sous-ensemble différent des partitions du topic.





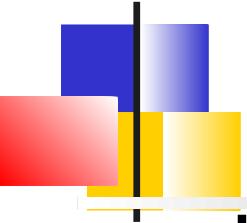
# Rééquilibrage dynamique des consommateurs

Lors de l'ajout d'un nouveau consommateur, celui-ci peut se faire affecter une partition consommée précédemment par un autre consommateur du groupe.

Lors de l'arrêt d'un consommateur, les partitions qui lui étaient assignées sont réaffectées à un autre consommateur

Cette répartition dynamique offre la scalabilité et la tolérance aux défaillances mais n'est pas spécialement désirable

- Durant le rééquilibrage les messages ne sont pas consommés
- les consommateurs si ils utilisent des caches sont obligés de les rafraîchir



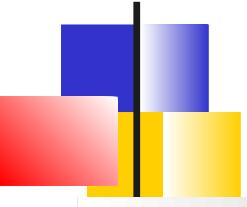
# Création de *KafkaConsumer*

---

L'instanciation d'un ***KafkaConsumer*** est similaire à celle d'une *KafkaProducer*

Plusieurs propriétés doivent être spécifiées dans une classe *Properties* :

- *bootstrap.servers*
- *key.deserializer* , et *value.deserializer*
- *group.id* qui spécifie le groupe de consommateur

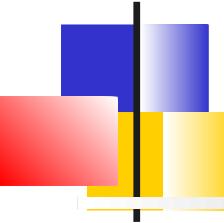


# Exemple

---

```
private Properties kafkaProps = new Properties();
kafkaProps.put("bootstrap.servers",
    "broker1:9092,broker2:9092");
kafkaProps.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
kafkaProps.put("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
kafkaProps.put("group.id", "myGroup");

consumer = new KafkaConsumer<String, String>(kafkaProps);
```



# Abonnement à un *topic*

---

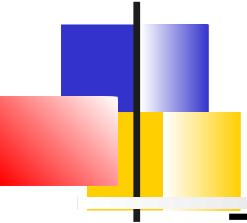
Un *KafkaConsumer* peut d'abonner à un ou plusieurs *topic(s)*

La méthode ***subscribe()*** prend une liste de *topics* ou une expression régulière.

Ex :

```
consumer.subscribe(Collections.singletonList("myTopic"));
```

```
consumer.subscribe("test.*");
```



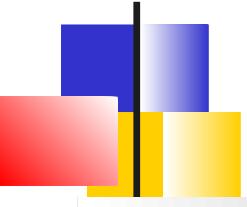
# Boucle de Polling

---

Typiquement, les consommateurs *poll* continuellement les *topics* auxquels ils sont abonnés.

Les objets retournés par *poll* sont une collection de ***ConsumerRecord*** qui encapsule :

- le message
- La partition
- L'offset
- Le timestamp



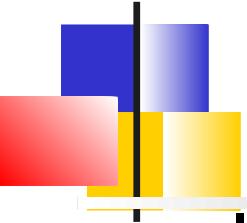
# Exemple

---

```
try {
    while (true) {
        // poll bloque pdt au maximum 100ms pour récupérer les messages
        // retourne immédiatement si des messages sont disponibles
        ConsumerRecords<String, String> records = consumer.poll(100);

        for (ConsumerRecord<String, String> record : records) {
            log.debug("topic = %s, partition = %s, offset = %d,
                      customer = %s, country = %s\n",
                      record.topic(),
                      record.partition(),
                      record.offset(), record.key(), record.value());

            // Traitement du record
            _handleRecord(record) ;
        }
    } finally {
        consumer.close();
}
```



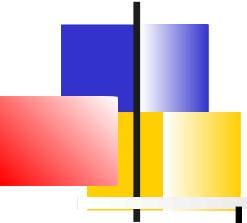
# Thread et consommateur

---

Il n'est pas possible d'avoir plusieurs consommateurs du même groupe dans la même thread et il n'est pas possible d'utiliser le même consommateur dans plusieurs threads

**=> 1 consommateur = 1 thread**

Pour exécuter plusieurs consommateurs d'un même groupe dans une application, il est utile d'utiliser les classes *ExecutorService* de Java ou de démarrer plusieurs processus.



# Offsets et Commits

---

Les consommateurs synchronisent l'état d'avancement de leur consommation, en périodiquement indiquant à Kafka le dernier offset traité.

Kafka appelle la mise à jour d'un offset : un **commit**

Pour committer, un consommateur envoie un message vers un *topic* particulier de Kafka :

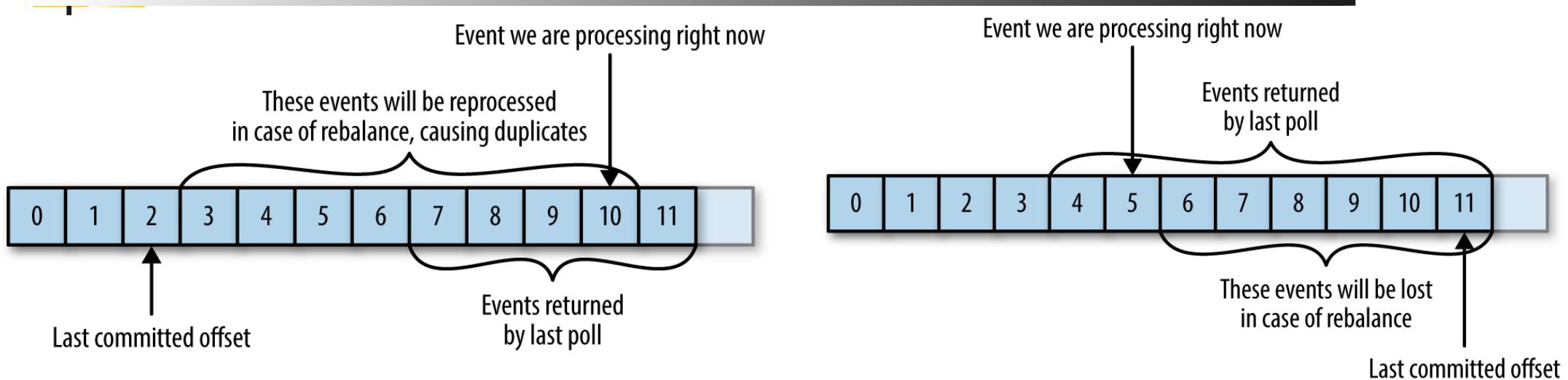
## **consumer\_offsets**

- Ce *topic* contient les offsets de chaque partition.

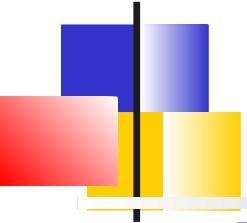
Lors d'une réaffectation de partitions, 2 risques doivent être pris en compte :

- Traiter 2 fois le même message
- Louper des messages

# Risques lors d'une réaffectation



Kafka propose plusieurs façons de gérer les commits

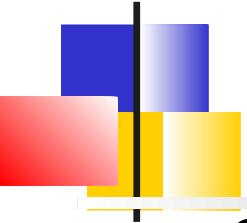


# Gestion des commits

---

Différentes alternatives sont possible pour gérer les commits :

- Laisser l'API Kafka committer automatiquement (défaut)
- Committer manuellement les commits
- Gérer les offsets en dehors de kafka et se positionner manuellement au bon offset lors d'une réaffectation de partition



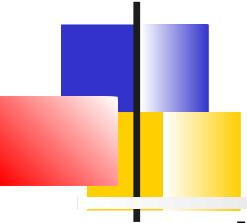
# Commit automatique

---

Si **`enable.auto.commit=true`** ,  
le consommateur valide automatiquement tout  
les **`auto.commit.interval.ms`** (par défaut  
5000), les plus grands offset reçus par `poll()`

=> Cette approche (simple) ne protège pas contre  
les duplications en cas de ré-affectation

Si le traitement des messages est asynchrone, on  
peut également perdre des messages



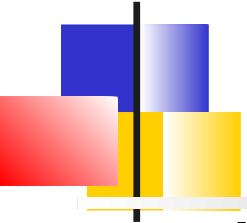
# Commit contrôlé

---

L'API Consumer permet de contrôler le moment du commit plutôt que de se baser sur un timer.

Si ***auto.commit.offset=false*** ,  
l'application doit explicitement committer les offsets

- Soit de façon bloquante avec ***commitSync()***
- Soit en asynchrone avec ***commitAsync()***



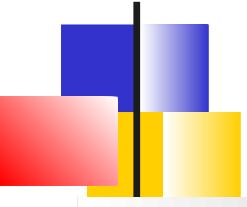
# Commit Synchrone

---

La méthode ***commitSync()*** valide les derniers offsets reçus par *poll()*

- La méthode est bloquante et retourne lorsque les offsets sont commités
- Elle lance une exception si un commit échoue

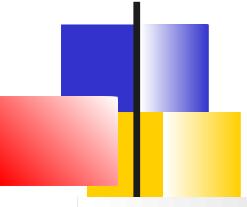
=> En cas de réaffectation, il y a toujours un risque pour que le messages soit traités plusieurs fois



# Exemple Java

---

```
while (true) {
    ConsumerRecords<String, String> records =
        consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        // Si réaffectation, possibilité de duplication
        log.info("topic = %s, partition = %s,
                  offset =%d, customer = %s, country = %s\n",
                  record.topic(), record.partition(),
                  record.offset(), record.key(), record.value());
    }
    try {
        consumer.commitSync();
    } catch (CommitFailedException e) {
        log.error("commit failed", e)
    }
}
```

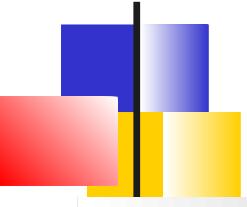


# Commit asynchrone

---

**commitAsync()** est non bloquante et il est possible de fournir un callback en argument

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        _handleRecord(record)
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition,
            OffsetAndMetadata> offsets, Exception exception) {
            if (e != null)
                log.error("Commit failed for offsets {}", offsets, e);
        }
    });
}
```

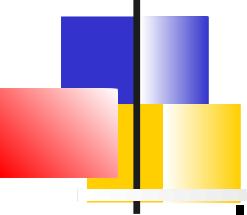


# Committer un offset spécifique

---

L'API permet également de fournir en argument une *Map* contenant les offsets que l'on veut valider pour chaque partition

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new  
    HashMap<>();  
int count = 0;  
....  
while (true) {  
    ConsumerRecords<String, String> records = consumer.poll(100);  
    for (ConsumerRecord<String, String> record : records) {  
        _handleRecord(record) ;  
        currentOffsets.put(new TopicPartition(record.topic(), record.partition()),  
            new OffsetAndMetadata(record.offset()+1, "no metadata"));  
        if (count % 1000 == 0)  
            consumer.commitAsync(currentOffsets, null);  
        count++;  
    }  
}
```



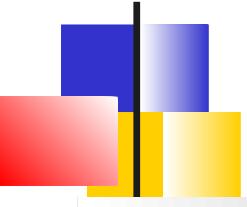
# Stocker les offsets hors de Kafka

---

Il n'y a pas d'obligation à stocker les offsets dans Kafka, un consommateur peut stocker lui même les offsets dans son propre data store.

- Si les offsets et le résultat du traitement sont stockés dans la même BD, on peut alors profiter d'une écriture transactionnelle.

Ce type de scénario permet d'obtenir facilement des garanties de livraison « *Exactly Once* » même en cas de défaillance



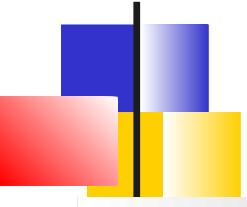
# Réagir aux réaffectations

---

Lors de réaffectations de partitions, les consommateurs peuvent être prévenus afin de prendre les mesures adéquates (commit, fermeture de ressources, ...)

Lors du *subscribe()*, il faut fournir une classe de type ***ConsumerRebalanceListener*** qui implémente 2 méthodes :

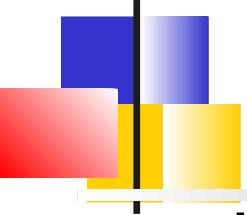
- `public void onPartitionsRevoked(  
    Collection<TopicPartition> partitions)`
- `public void onPartitionsAssigned(  
    Collection<TopicPartition> partitions)`



# Exemple

---

```
private class HandleRebalance implements  
ConsumerRebalanceListener {  
  
    public void onPartitionsAssigned(  
        Collection<TopicPartition> partitions) { }  
  
    public void onPartitionsRevoked(  
        Collection<TopicPartition> partitions) {  
        log.info("Lost partitions in rebalance.  
            Committing current offsets:" + currentOffsets);  
        consumer.commitSync(currentOffsets);  
    }  
}
```

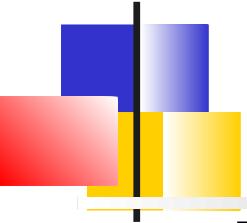


# Consommation de messages avec des offsets spécifiques

---

L'API de consommation permet d'indiquer un offset spécifique :

- ***seekToBeginning(TopicPartition tp)*** :  
Revenir au début de la partition
- ***seekToEnd(TopicPartition tp)*** :  
Se placer à la fin
- ***seek(TopicPartition, long)*** :  
Se placer à un offset particulier  
Cela permet de stocker les offsets en dehors de Kafka et en cas d'erreur ou redémarrage repartir à partir des offsets sauvegardés



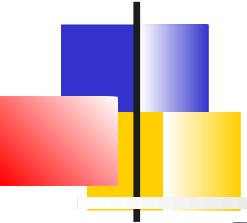
# Sortie de boucle

---

Pour sortir de la boucle de *poll*, il faut qu'une autre thread appelle ***consumer.wakeup()*** qui a pour effet de lancer une *WakeupException* lors de l'appel à *poll*.

Le consommateur doit alors faire un appel explicite à *close()*

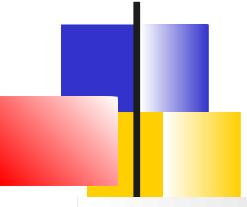
On peut utiliser  
*Runtime.addShutdownHook(Thread hook)*



# Exemple

---

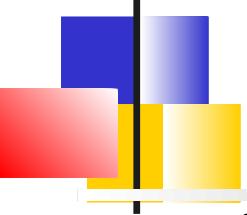
```
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {
        System.out.println("Starting exit...");
        consumer.wakeup();
        try {
            mainThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});
```



# Exemple (2)

---

```
try {
    // looping until ctrl-c, the shutdown hook will cleanup on exit
    while (true) {
        ConsumerRecords<String, String> records =
            movingAvg.consumer.poll(1000);
        log.info(System.currentTimeMillis() + "-- waiting for data...");
        for (ConsumerRecord<String, String> record : records) {
            log.info("offset = %d, key = %s,value = %s\n",
                    record.offset(), record.key(), record.value());
        }
        for (TopicPartition tp: consumer.assignment())
            log.info("Committing offset
atposition:"+consumer.position(tp));
        movingAvg.consumer.commitSync();
    }
} catch (WakeupException e) {
} finally {
    consumer.close();
    log.info("Closed consumer and we are done");
}
```



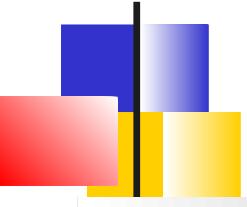
# Affectation statique des partitions

---

Si l'application ne comporte que des consommateurs dédiés statiquement à des partitions, alors la fonctionnalité de groupe n'est pas nécessaire.

Dans ce cas, l'application assigne explicitement les partitions à ses consommateurs.

L'API ***assign()*** est alors utilisée à la place de *subscribe()*



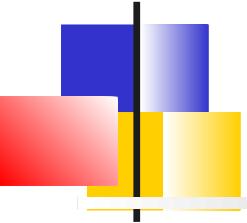
# Exemple

---

```
List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic");

if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new TopicPartition(partition.topic(), partition.partition()));

consumer.assign(partitions);
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(1000);
    for (ConsumerRecord<String, String> record: records) {
        log.info("topic = %s, partition = %s, offset = %d,
                 customer = %s, country = %s\n",
                 record.topic(), record.partition(), record.offset(),
                 record.key(), record.value());
    }
    consumer.commitSync();
}
}
```

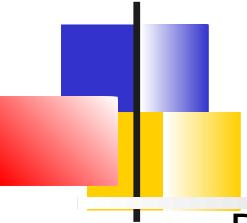


## *auto.offset.reset*

---

Si le consommateur n'a pas d'offset défini dans Kafka, le comportement est piloté par la propriété ***auto.offset.reset*** :

- *latest* (défaut), l'offset est initialisé au dernier offset
- *earliest* : L'offset est initialisé au premier offset disponible

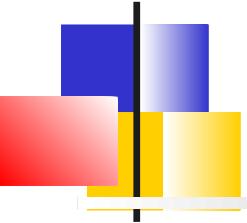


# Autres propriétés

---

## D'autres propriétés

- **fetch.min.bytes** : Volume minimum de données à recevoir. Permet de réduire la charge sur le broker et le consommateur
- **fetch.max.wait.ms** : Attente maximale avant de récupérer les données
- **max.partition.fetch.bytes** : Maximum de données par partition ramenées lors d'un poll. Par défaut 1Mo
- **max.poll.records** : Maximum de record via un *poll()*
- **session.timeout.ms** : Le temps faisant expirer la session et déclarer le consommateur comme down. Par défaut 10s
- **heartbeat.interval.ms** : L'intervalle d'envoi des heartbeat. Par défaut 3s
- **partition.assignment.strategy** : Stratégie d'affectation des partitions *Range* (défaut), *RoundRobin* ou *Custom*
- **client.id** : Une chaîne de caractère utilisé pour les métriques.
- **receive.buffer.bytes** et **send.buffer.bytes** : Taille des buffers TCP

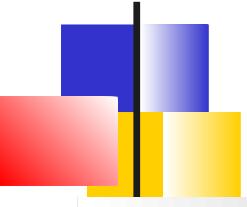


# Apache Kafka et ses APIs

---

Les APIs Kafka  
Producer API  
Consumer API

**Sérialisation et Schema Registry**  
Autres APIs



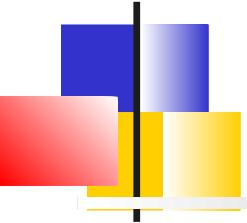
# Introduction

---

Lors d'évolution des applications, le format des messages est susceptible de changer.

=> Afin de s'assurer que ses évolutions ne génèrent pas de problème chez les consommateurs, il est nécessaire d'utiliser un gestionnaire de schéma capable de détecter les problèmes de compatibilité.

C'est le rôle de ***Schema Confluent Registry*** qui supporte les formats de sérialisation JSON, Avro et ProtoBuf

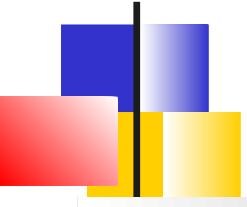


# Apache Avro

---

*Apache Avro est un système de sérialisation de données.*

- Il utilise une structure JSON pour définir le **schéma**, permettant la sérialisation entre les octets et les données structurées.
- Les outils associés à Avro sont capables de générer les classes Java<sup>1</sup> correspondantes au schéma.
- Il offrent également un format de sérialisation binaire plus compact



# Utilisation du schéma

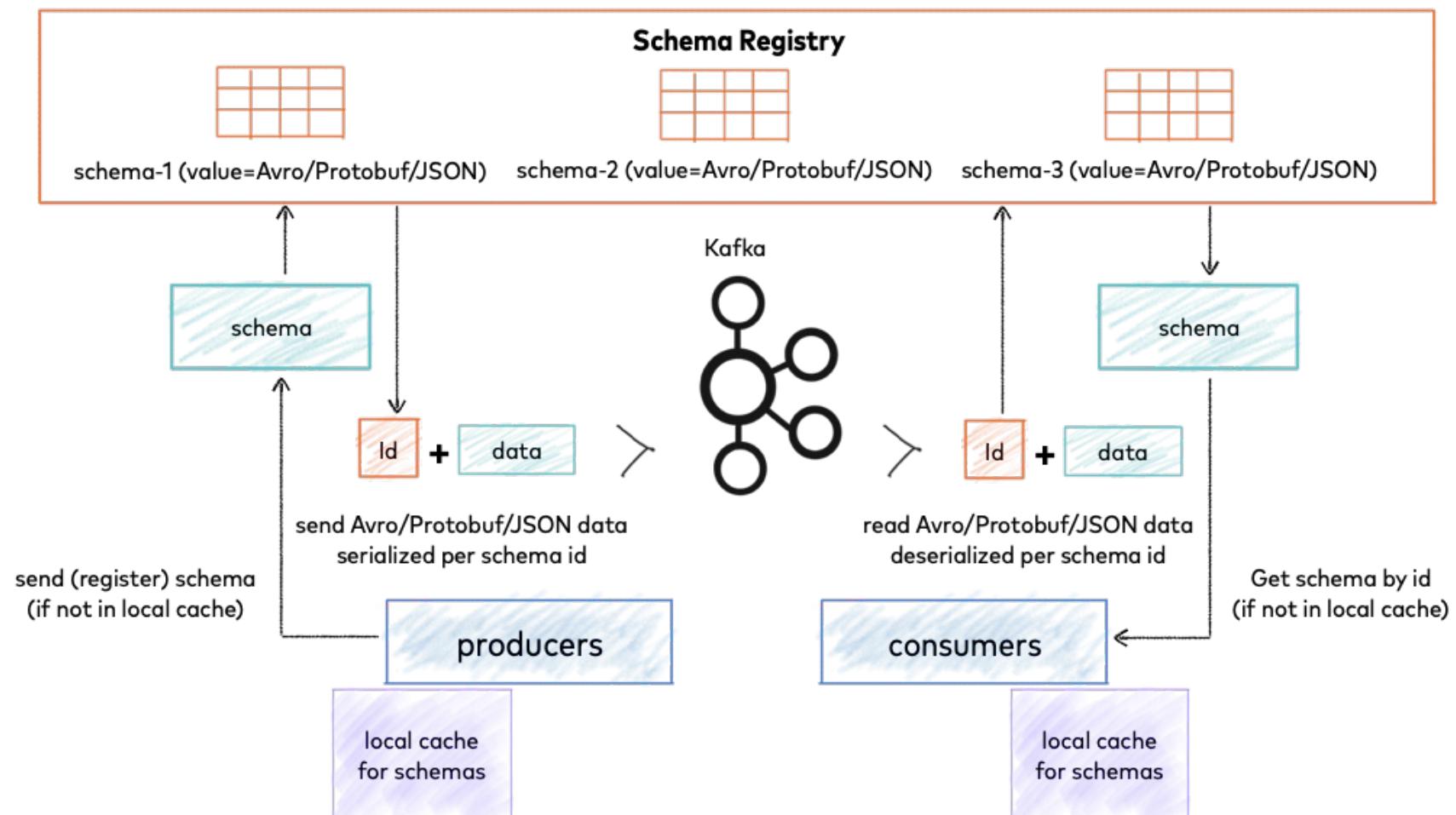
---

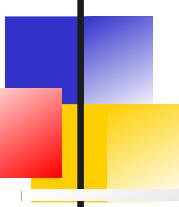
**Confluent Schema Registry** offre une API Rest

- permettant de stocker des Schema
- De détecter les compatibilité entre schémas  
Si le schéma est incompatible, le producteur est empêché de produire vers le topic  
Il faudra publier une autre version vers un autre topic

Les sérialiseurs inclut l'id du schéma dans les messages

# Schema Registry





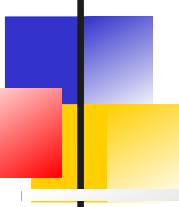
# Modes de compatibilité

---

Trois niveaux de compatibilité sont pris en charge par *Schema Registry* :

- **Compatibilité backward** : Tous les messages de la version précédente du schéma sont également valides selon la nouvelle version
- **Compatibilité forward** : Tous les messages de la nouvelle version sont également valides selon la version précédente du schéma
- **Compatibilité full** : La version précédente du schéma et la nouvelle version sont toutes deux compatibles ascendante et descendante

Les vérifications de compatibilité s'effectuent soit juste avec la précédente version, soit avec toutes les versions précédentes.



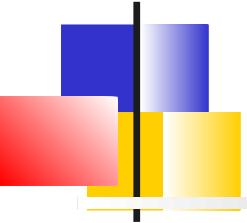
# Choix du mode

---

La compatibilité backward, configuration par défaut, est à privilégier.

- Elle permet à un consommateur de lire les anciens messages, ce qui est compatible avec la possibilité de Kafka de s'abonner à posteriori.
- Cependant, cela oblige à upgrader les consommateurs en premier afin qu'il s'adapte à la nouvelle version du message.

Avec la compatibilité forward, ce sont les producteurs qui sont mis à jour en premier. Par contre, les possibilités de mises à jour sont très restrictives pour le producteur.



# Enregistrement de schéma

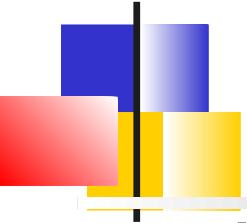
---

L'enregistrement du schéma peut s'effectuer via l'outil de build :

```
mvn schema-registry:register
```

Ou en utilisant la librairie cliente Java de *Schema Registry* :

```
// Lecture du schéma
Schema avroSchema = new Schema.Parser().parse(inputStream);
// Instanciation du client
CachedSchemaRegistryClient client = new CachedSchemaRegistryClient(REGISTRY_URL, 20);
// Enregistrement du sujet, le nom correspond au nom du topic + suffixe
client.register(topicName +"-value", new AvroSchema(avroSchema);
```

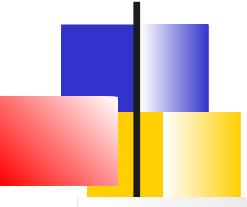


# Producteur

---

Les propriétés du producteur Kafka doivent contenir :

- ***schema.registry.url*** :  
L'adresse du serveur de registry
- Le sérialiseur de valeur :  
***io.confluent.kafka.serializers.KafkaAvroSerializer***

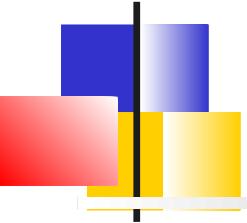


# Consommateur

*KafkaConsumer* doit également préciser l'URL et le déserialiseur à  
***io.confluent.kafka.serializers.KafkaAvroDeserializer***

Il peut récupérer les messages sous la forme de ***GenericRecord*** plutôt que des classes spécialisés.

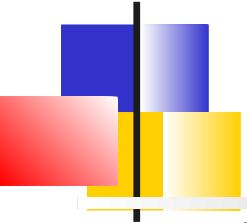
```
ConsumerRecords<String, GenericRecord> records =  
    consumer.poll(Duration.ofMillis(sleep));  
for (ConsumerRecord<String, GenericRecord> record : records) {  
    System.out.println("Value is " + record.value());
```



# Apache Kafka et ses APIs

---

Les APIs Kafka  
Producer API  
Consumer API  
Sérialisation et Schema Registry  
**Autres APIs**

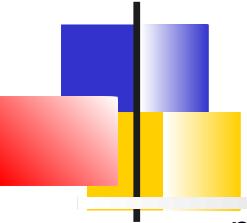


# Introduction

---

Kafka propose 2 autres APIs :

- **Admin API** : Client d'administration permettant de gérer et inspecter les topics, brokers, configurations et ACLs
- **Streams API** : Librairie cliente pour faciliter le traitement du flux dont les entrées/sorties sont des *topics* Kafka

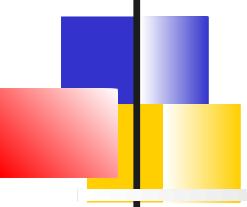


# Exemple Admin : Lister les configurations

---

```
public class ListingConfigs {

    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
        Properties config = new Properties();
        config.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        AdminClient admin = AdminClient.create(config);
        for (Node node : admin.describeCluster().nodes().get()) {
            System.out.println("-- node: " + node.id() + " --");
            ConfigResource cr = new ConfigResource(ConfigResource.Type.BROKER,
                "0");
            DescribeConfigsResult dcr =
                admin.describeConfigs(Collections.singleton(cr));
            dcr.all().get().forEach((k, c) -> {
                c.entries()
                    .forEach(configEntry -> {
                        System.out.println(configEntry.name() + " = " +
                            configEntry.value());
                    });
            });
        }
    }
}
```

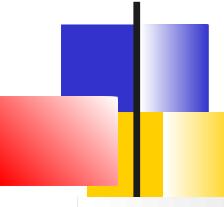


# Exemple Admin

## Créer un topic

---

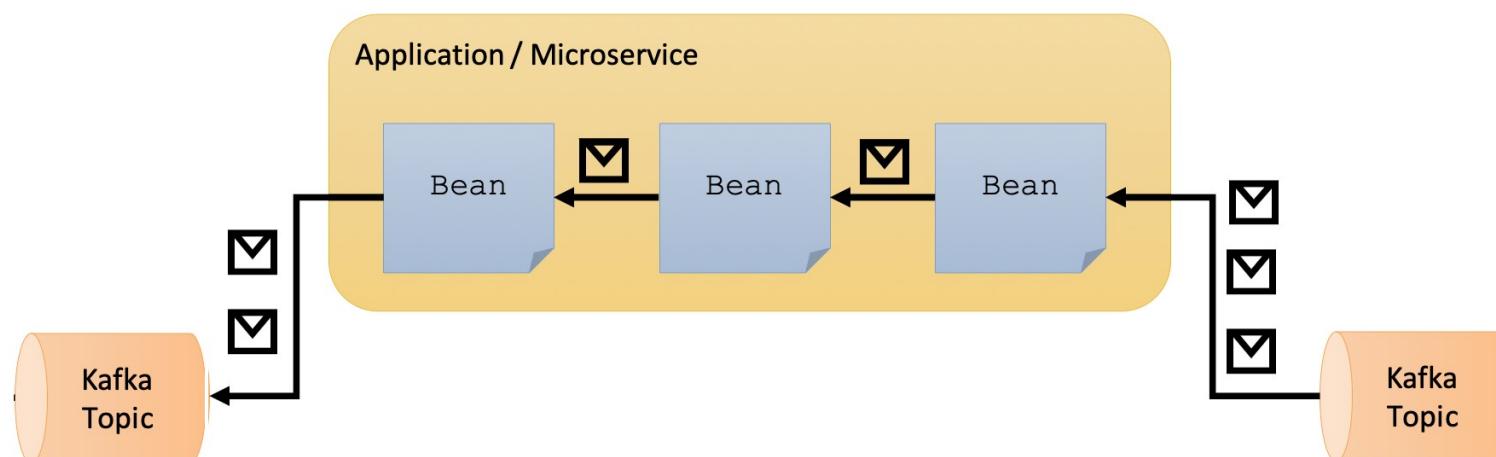
```
public class CreateTopic {  
    public static void main(String[] args) throws ExecutionException,  
        InterruptedException {  
        Properties config = new Properties();  
        config.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,  
        "localhost:9092");  
        AdminClient admin = AdminClient.create(config);  
        //Créer un nouveau topics  
        System.out.println("-- creating --");  
        NewTopic newTopic = new NewTopic("my-new-topic", 1, (short) 1);  
        admin.createTopics(Collections.singleton(newTopic));  
  
        //lister  
        System.out.println("-- listing --");  
        admin.listTopics().names().get().forEach(System.out::println);  
    }  
}
```

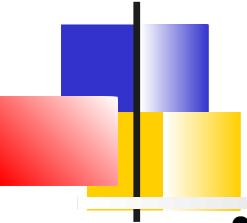


# Kafka Streams

---

**Kafka Streams API** est une bibliothèque cliente pour la création d'applications et de micro-services, où les données d'entrée et de sortie sont stockées dans des topics Kafka

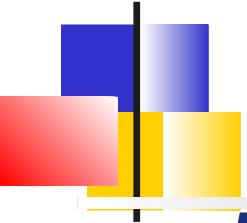




# Apports de *KafkaStream*

---

- Librairie simple et légère, facilement intégrable, seule dépendance celle d'Apache Kafka.
- Abstractions ***KStream*** et ***KTable*** permettant la manipulation de flux d'évènements, transformation, filtrage, jointures et agrégations
- Peut garantir que chaque évènement soit traité une et une seule fois, même en cas de défaillance.
- Temps de latence des traitements en millisecondes,
- Supporte des opérations de fenêtrage temporel avec l'arrivée des événements dans le désordre.
- Définition d'une topologie de processeurs d'évènements sous forme de DSL ou programmatiquement.



# *KStream*

---

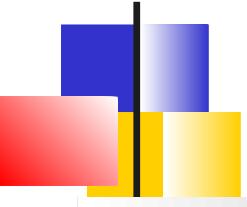
***KStream*** est une abstraction représentant un flux de données illimité dont:

- Les événements sont ordonnés
- Les événements sont immuables
- On peut rejouer un flux d'événements

*KStream* offre un grand nombre d'opérateurs de manipulation d'événements qui produise des déclinaisons de *KStream*

Un *KStream* peut

- être construit à partir d'un topic Kafka
- Être la source d'un topic

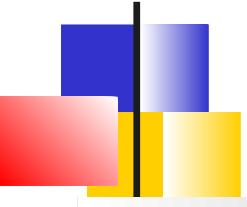


# KStream et KTable

---

L'abstraction ***KTable*** représente un ensemble de fait qui évoluent, chaque enregistrement est associé à une clé.

- Une *KTable* peut être construit à partir d'un *Kstream*.  
Seule la dernière valeur ou une agrégation, pour une clé donnée, est conservée
- *KTable* peut être transformé en *Kstream*.  
Le flux d'évenement correspond aux flux des modifications des entités



# Application KafkaStream

---

Une application KafkaStream définit sa logique de traitement à travers une ou plusieurs **topologies de processeurs**, i.e. un graphe de processeurs de flux connectés

Un processeur représente une étape de traitement qui prend en entrée un évènement, le transforme puis produit un ou plusieurs événements

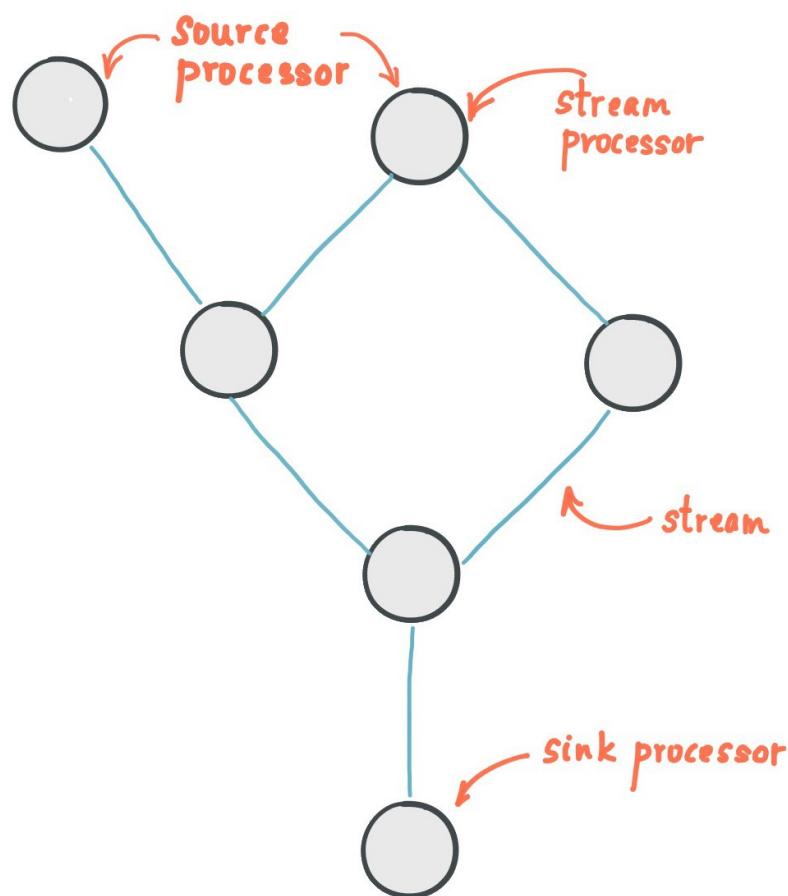
Certains processeurs :

- n'ont pas de connexions entrantes : **Source**
- d'autres n'ont pas de connexions sortantes : **Sink**
- Les autres ont des connexions entrantes et sortantes

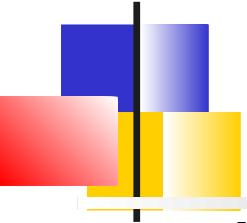
La topologie peut être spécifiée programmatiquement ou par un DSL

Certains processeurs sont stateful et stockent un état dans un **StateStore** (Topic Kafka intermédiaire)

# Topologie processeurs



PROCESSOR TOPOLOGY



# Scalabilité

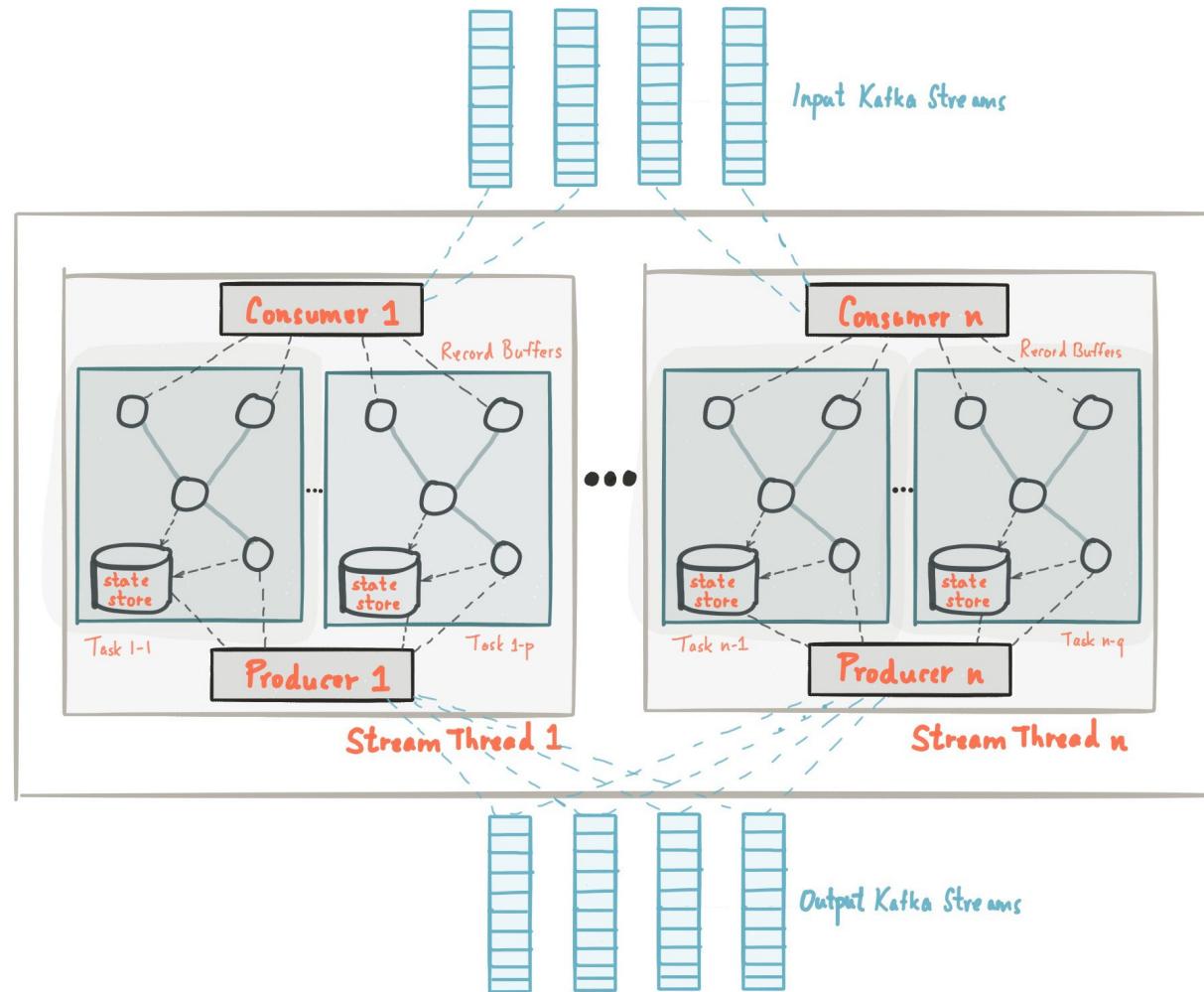
---

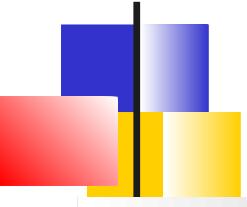
KafkaStream offre plusieurs moyens pour la scalabilité

- Démarrage de n-processus
- Définition dans un processus de Task  
(correspondant à des pools de threads)

Bien sûr, le degré de parallélisme est fixé par le nombre de partitions du topic entrant

# Architecture et scalabilité





# Exemple

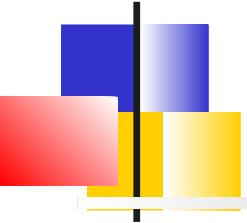
---

```
// Propriétés : ID, BOOTSTRAP, Sérialiseur/Désérialiseur
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-pipe");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());

// Création d'une topologie de processeurs
final StreamsBuilder builder = new StreamsBuilder();
builder.<String, String>stream("streams-plaintext-input")
    .flatMapValues(value -> Arrays.asList(value.split("\\w+")))
    .to("streams-linesplit-output");

final Topology topology = builder.build();

// Instanciation du Stream à partir d'une topologie et des propriétés
final KafkaStreams streams = new KafkaStreams(topology, props);
```



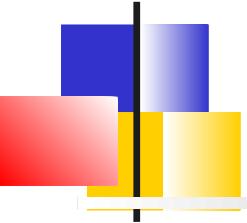
# Exemple (2)

---

```
final CountDownLatch latch = new CountDownLatch(1);

// attach shutdown handler to catch control-c
Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-hook") {
    @Override
    public void run() {
        streams.close();
        latch.countDown();
    }
});

// Démarrage du stream
try {
    streams.start();
    latch.await();
} catch (Throwable e) {
    System.exit(1);
}
System.exit(0);
```



# Spring-Kafka

---

## Introduction

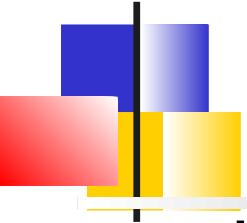
Production de message

Consommation

Transaction

Sérialisation / Désérialisation

Traitement des exceptions

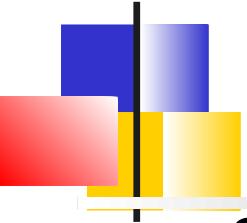


# Introduction

---

Le starter *SpringKafka* apporte :

- Les concepts Spring Ioc et DI
- Les auto-configurations SpringBoot
- Des classes template pour envoyer les messages
- **@KafkaListener** sur des POJOs pour recevoir les messages
- Un similarités avec les autres intégrations de *MessageBroker*, en particulier RabbitMQ et JMS



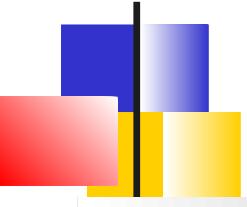
# Connexions à Kafka

---

Spring permet 3 types de clients Kafka

- **KafkaAdmin** : Utilisé principalement pour créer et configurer les topics
- **ProducerFactory** : Permettant de configurer la production de messages
- **ConsumerFactory** : Permettant de configurer la consommation

SpringBoot configure automatiquement un instance de ces classes en s'appuyant sur les propriétés de configuration

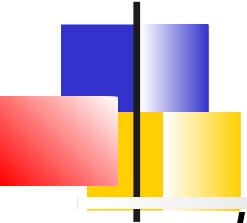


# Configuration des topics

---

Si un bean *KafkaAdmin* est disponible (automatique avec SpringBoot), il est possible de définir des beans **NewTopic** qui permettent la création automatique de topic

```
@Bean
public NewTopic topic2() {
    return TopicBuilder.name("thing2")
        .partitions(10)
        .replicas(3)
        .config(TopicConfig.COMPRESSION_TYPE_CONFIG, "zstd")
        .build();
}
```



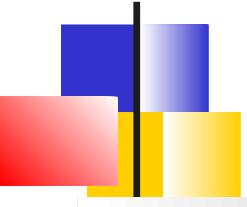
# KafkaAdmin et AdminClient

---

*KafkaAdmin* fournit les méthodes pour créer et examiner des topics : *createOrModifyTopics()*, *describeTopics()*

On peut également accéder directement à la classe *Kafka AdminClient* pour des fonctionnalités plus avancées

```
@Autowired  
private KafkaAdmin admin;  
  
...  
AdminClient client =  
    AdminClient.create(admin.getConfigurationProperties());  
...  
client.close()
```

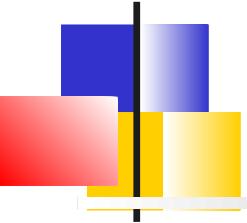


# Listeners

---

*Les factories pour les producteurs et les consommateurs de messages peuvent être configurés avec des Listeners*

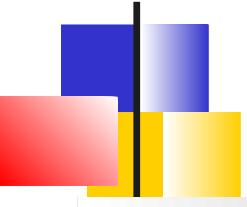
```
interface Listener<K, V> {  
    default void producerAdded(String id, Producer<K, V> producer) {}  
    default void producerRemoved(String id, Producer<K, V> producer) {}  
}  
  
interface Listener<K, V> {  
  
    default void consumerAdded(String id, Consumer<K, V> consumer) {}  
    default void consumerRemoved(String id, Consumer<K, V> consumer) {}  
}
```



# Spring-Kafka

---

Introduction  
**Production de message**  
Consommation  
Transaction  
Sérialisation / Désérialisation  
Traitement des exceptions



# Production de messages

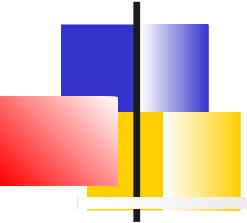
---

*ProducerFactory* est utilisé pour centraliser la configuration de la production. Elle permet de construire des classes \*Template qui encapsule un Producer Kafka

- L'implémentation la plus utilisée est  
***DefaultKafkaProducerFactory*** qui se configure avec une Map

Différents objets *Template* sont fournis pour l'émission de messages

- ***KafkaTemplate*** encapsulant un *KafkaProducer*
- ***RoutingKafkaTemplate*** permet de sélectionner un *KafkaProducer* en fonction du nom du *Topic*
- ***ReplyingKafkaTemplate*** permettant des interactions Request/Response



# Configuration sans SpringBoot

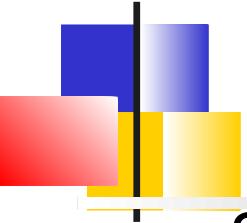
---

```
@Bean
public ProducerFactory<Integer, String> producerFactory() {
    return new DefaultKafkaProducerFactory<>(producerConfigs());
}

@Bean
public Map<String, Object> producerConfigs() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class);
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);

    return props;
}

@Bean
public KafkaTemplate<Integer, String> kafkaTemplate() {
    return new KafkaTemplate<Integer, String>(producerFactory());
}
```



# Avec SpringBoot

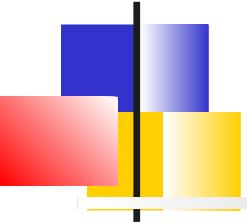
---

SpringBoot configure automatiquement un *ProducerFactory* à partir des propriétés trouvées dans *application.yml*

```
spring:  
  kafka:  
    bootstrap-servers:  
      - localhost:9092  
    producer:  
      value-serializer: org.apache.kafka.common.serialization.StringSerializer  
      key-serializer: org.apache.kafka.common.serialization.IntegerSerializer
```

Il suffit alors de s'injecter un *KafkaTemplate* (singleton par défaut partagé par toutes les threads)

```
@Autowired  
KafkaTemplate<Integer, String> kafkaTemplate
```



# *ProducerFactory*

---

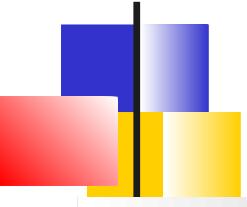
Par défaut, *DefaultKafkaProducerFactory* crée un producer singleton

- La propriété *producerPerThread* permet de créer un Producer par thread

Il peut également être utile d'accéder à *ProducerFactory* pour mettre à jour dynamiquement sa configuration via :

- `updateConfigs(Map<String, Object> updates)`
- `void removeConfig(String configKey);`

Utile pour les rotations de clés par exemple



# Méthodes d'envoi

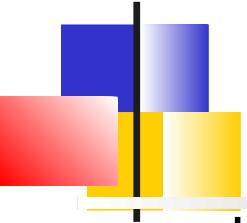
---

KafkaTemplate propose 2 méthodes d'envoi retournant un **CompletableFuture<SendResult>** :

- **sendDefault** qui nécessite d'avoir défini un topic par défaut
- **send** où le topic est fourni en argument

Différentes signatures sont possibles pour ces 2 méthodes :

- Juste la donnée
- La donnée et la clé
- La donnée, la clé et le timestamp
- *ProducerRecord<K, V>* record de Kafka;
- *Message<?>* message de spring-integration



# Réponse à l'envoi

---

La réponse est encapsulée dans un *CompletableFuture* permettant différents types d'interaction

- Fire and Forget
- Synchrone

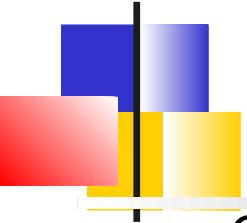
```
CompletableFuture<SendResult<Integer, String>> future = template.send("myTopic", "something");  
SendResult result = future.get();
```

- Asynchrone avec call-back

```
CompletableFuture<SendResult<Integer, String>> future = template.send("myTopic", "something");  
future.whenComplete((result, ex) -> {  
    ...  
});
```

*SendResult* encapsule les classes de Kafka

- *ProducerRecord* : L'enregistrement envoyé
- *RecordMetadata* : Partition, offset, timestamp, objets sérialisés



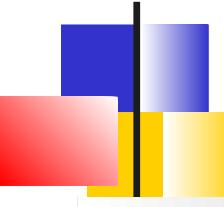
# *ProducerListener*

On peut également associer un ***ProducerListener*** au template via sa méthode *setProducerListener()*

Les méthodes de call-back seront alors appelées pour tous les envois du Template

```
public interface ProducerListener<K, V> {  
    void onSuccess(ProducerRecord<K, V> producerRecord, RecordMetadata recordMetadata);  
  
    void onError(ProducerRecord<K, V> producerRecord, RecordMetadata recordMetadata, Exception exception);  
}
```

Par défaut, KafkaTemplate est configuré avec un *LoggingProducerListener*, qui trace les erreurs et ne fait rien lorsque l'envoi réussit.



# RoutingKafkaTemplate

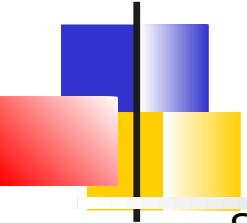
---

**RoutingKafkaTemplate** permet de sélectionner le producer au moment de l'exécution en fonction du nom du topic

```
@Bean
public RoutingKafkaTemplate routingTemplate(GenericApplicationContext context, ProducerFactory<Object, Object>
    pf) {
    // Cloner le ProducerFactory par défaut avec un sérialiseur différent
    Map<String, Object> configs = new HashMap<>(pf.getConfigurationProperties());
    configs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, ByteArraySerializer.class);
    DefaultKafkaProducerFactory<Object, Object> bytesPF = new DefaultKafkaProducerFactory<>(configs);
    context.registerBean("bytesPF", DefaultKafkaProducerFactory.class, bytesPF);

    Map<Pattern, ProducerFactory<Object, Object>> map = new LinkedHashMap<>();
    map.put(Pattern.compile("two"), bytesPF); // Topic « two » utilise le ByteSerializer
    map.put(Pattern.compile("."), pf); // Défaut : StringSerializer
    return new RoutingKafkaTemplate(map);
}

@Bean
public ApplicationRunner runner(RoutingKafkaTemplate routingTemplate) {
    return args -> {
        routingTemplate.send("one", "thing1");
        routingTemplate.send("two", "thing2".getBytes());
    };
}
```



# *ReplyingKafkaTemplate*

---

Sous-classe de *KafkaTemplate* permettant un mode requête/réponse :

- Un message est envoyé sur un topic *request*
- Un consommateur répond sur un topic *Response*

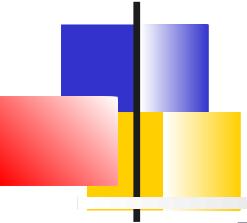
Lors de son instantiation, un *GenericMessageListenerContainer* représentant un consommateur du topic *Response* est fournie.

L'envoi de message peut alors se faire via une des 2 méthodes supplémentaires :

```
RequestReplyFuture<K, V, R> sendAndReceive(ProducerRecord<K, V> record);  
RequestReplyFuture<K, V, R> sendAndReceive(ProducerRecord<K, V> record, Duration  
replyTimeout);
```

La valeur de retour est un *CompletableFuture* renseigné de façon asynchrone qui contient :

- La clé
- La valeur envoyée
- La réponse du consommateur

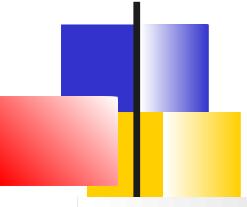


# Entêtes

---

Par défaut, 3 en-têtes sont utilisés :

- KafkaHeaders.CORRELATION\_ID - utilisé pour corréler la réponse à une requête
- KafkaHeaders.REPLY\_TOPIC - utilisé pour indiquer au serveur où répondre
- KafkaHeaders.REPLY\_PARTITION - (facultatif) utilisé pour indiquer au serveur à quelle partition répondre



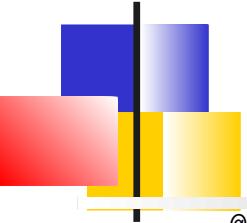
# Exemple (1)

## Création des beans

---

```
@Bean
public ReplyingKafkaTemplate<String, String, String> replyingTemplate(
    ProducerFactory<String, String> pf,
    ConcurrentMessageListenerContainer<String, String> repliesContainer) {
    return new ReplyingKafkaTemplate<>(pf, repliesContainer);
}
```

```
// Possibilité également de s'appuyer sur l'autoconfiguration SpringBoot
@Bean
public ConcurrentMessageListenerContainer<String, String> repliesContainer(
    ConcurrentKafkaListenerContainerFactory<String, String> containerFactory)
{
    ConcurrentMessageListenerContainer<String, String> repliesContainer =
        containerFactory.createContainer("kReplies");
    repliesContainer.getContainerProperties().setGroupId("repliesGroup");
    repliesContainer.setAutoStartup(false);
    return repliesContainer;
}
```

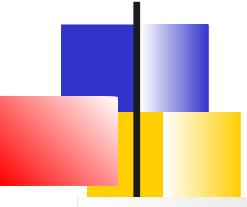


# Exemple (2)

## Envoi de message

---

```
@Bean
public ApplicationRunner runner(ReplyingKafkaTemplate<String, String, String> template) {
    return args -> {
        if (!template.waitForAssignment(Duration.ofSeconds(10))) {
            throw new IllegalStateException("Reply container did not initialize");
        }
        ProducerRecord<String, String> record = new ProducerRecord<>("kRequests", "foo");
        RequestReplyFuture<String, String> replyFuture = template.sendAndReceive(record);
// Vérification de l'envoi de la requête
        SendResult<String, String> sendResult = replyFuture.getSendFuture().get(10,
                TimeUnit.SECONDS);
        System.out.println("Résultat de l'envoi ok: " + sendResult.getRecordMetadata());
// Lecture de la valeur de la réponse
        ConsumerRecord<String, String> consumerRecord = replyFuture.get(10, TimeUnit.SECONDS);
        System.out.println("Valeur de la réponse : " + consumerRecord.value());
    };
}
```



# Exemple (3)

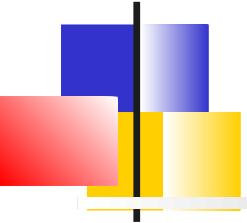
---

```
@KafkaListener(id="server", topics = "kRequests")
@SendTo
public String listen(String in) {
    System.out.println("Server received: " + in);
    return in.toUpperCase();
}
```

Le topic de réponse est indiqué dans l'entête

*KafkaHeaders.REPLY\_TOPIC*.

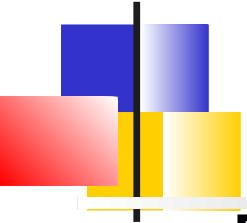
L'annotation `@SendTo` a pour effet de renvoyer sur le topic de réponse la valeur renvoyée par la méthode



# Spring-Kafka

---

Introduction  
Production de message  
**Consommation**  
Transaction  
Sérialisation / Désérialisation  
Traitement des exceptions



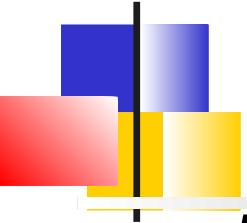
# Introduction

---

Pour consommer des messages, il est nécessaire de définir :

- **MessageListenerContainer** : Définit le modèle de threads des listeners
- **MessageListener** : Mode de réception des messages

L'annotation **@KafkaListener** sur une méthode permet d'associer une méthode d'un POJO à un *MessageListener*



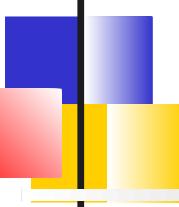
# *MessageListener*

---

*MessageListener* est l'interface permettant de consommer les *ConsumerRecords* de Kafka retournés par la boucle *poll*

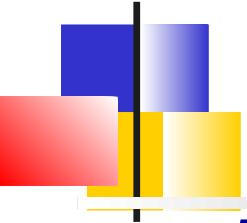
Il offre différentes alternatives :

- Traitement individuel ou par lot
- Commit géré par le container ou manuel via la classe **Acknowledgment**
- Accès au *KafkaConsumer* sous-jacent



# Interfaces MessageListener

```
// Traitement individuel des ConsumerRecord retournée par poll() de Kafka :  
// commit gérées par le container  
MessageListener<K, V> : onMessage(ConsumerRecord<K, V> );  
ConsumerAwareMessageListener<K, V> : onMessage(ConsumerRecord<K, V>,  
Consumer<?, ?> ); }  
  
// Traitement individuel avec commit manuel (Acknowledgment)  
AcknowledgingMessageListener<K, V> : onMessage(ConsumerRecord<K, V>,  
Acknowledgment);  
AcknowledgingConsumerAwareMessageListener<K, V> : onMessage(ConsumerRecord<K,  
V>, Acknowledgment, Consumer<?, ?>);  
  
// Traitement par lot avec commit géré par container  
BatchMessageListener<K, V> : onMessage(List<ConsumerRecord<K, V>>);  
BatchConsumerAwareMessageListener<K, V> : onMessage(List<ConsumerRecord<K, V>>,  
Consumer<?, ?>);  
  
// Traitement par lot avec commit manuel  
BatchAcknowledgingMessageListener<K, V> : onMessage(List<ConsumerRecord<K, V>>,  
Acknowledgment);  
BatchAcknowledgingConsumerAwareMessageListener<K, V> :  
onMessage(List<ConsumerRecord<K, V>>, Acknowledgment, Consumer<?, ?>); }
```



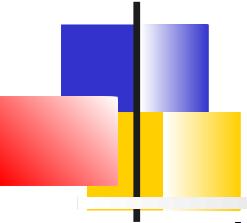
# **MessageListenerContainer**

---

**MessageListenerContainer** contient la configuration de la connexion à Kafka, et les topics ou partitions à consommer.  
Il effectue la boucle de *poll()*

2 implémentations sont fournies :

- **KafkaMessageListenerContainer** :  
Implémentation mono-thread
- **ConcurrentMessageListenerContainer** :  
Implémentation multi-thread permettant d'ajuster le nombre de threads au nombre de partitions du Topic



# Instanciation

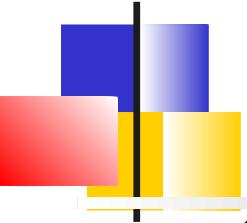
---

- Le constructeur des 2 implémentations de *MessageListenerContainer* nécessite 2 arguments :
- ***ConsumerFactory<K, V>*** : Encapsule les propriétés Kafka
  - ***ContainerProperties*** : Encapsule les informations sur les topics ou partition à consommer

Constructeur *ContainerProperties* :

```
public ContainerProperties(TopicPartitionOffset... topicPartitions)
public ContainerProperties(String... topics)
public ContainerProperties(Pattern topicPattern)
```

*ContainerProperties* propose également la méthode *setMessageListener()* permettant d'associer le listener au Container

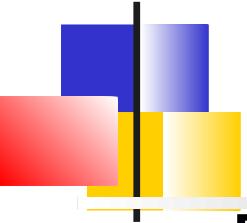


# *ConcurrentMessageListener*

---

*ConcurrentMessageListenerContainer* a une propriété **concurrency** supplémentaire déterminant le degré de multi-threading :

- A chaque thread est associé une propriété **client-id = <prefixe>-<n>**
- A chaque thread est associé 1 ou plusieurs partitions avec la distribution par défaut de Kafka
  - Peut être influencé par la propriété :  
*spring.kafka.consumer.properties.partition.assignment.strategy*

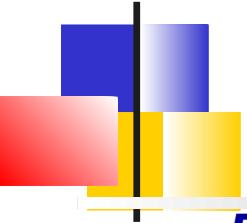


# Commits

---

Différentes options pour la gestion des commits :

- Si **`enable.auto.commit = true`**. On s'appuie sur les commits automatiques de Kafka
- Sinon (défaut), on s'appuie sur les commits Spring via l'énumération **`AckMode`**  
Il propose de nombreuses options qui influent sur le nombre de commits et donc sur des compromis entre performance et risque



# AckMode

---

**RECORD** : Commit au retour du listener => à chaque enregistrement

**BATCH (Défaut)** : Commit lorsque tous les enregistrements renvoyés par poll() ont été traités.

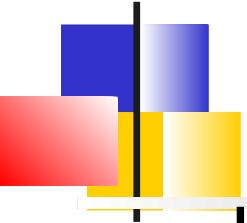
**TIME** : Commit l'offset du dernier poll si *ackTime* a été dépassé depuis la dernière validation.

**COUNT** : Commit l'offset du dernier poll si *ackCount* messages ont été reçus depuis la dernière validation.

**COUNT\_TIME** : similaire à TIME et COUNT, le commit est effectué si l'une ou l'autre des conditions est vraie.

**MANUAL** : Le MessageListener est responsable d'effectuer le commit via la méthode *Acknowledgement.acknowledge()*. Le commit vers Kafka est envoyé avec BATCH

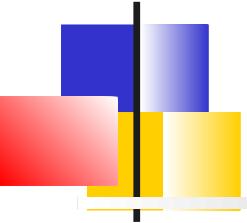
**MANUAL\_IMMEDIATE** : Commit effectué lorsque la méthode *Acknowledgement.acknowledge()* est appelée par le *messageListener*.



# Acknowledgment

L'interface **Acknowledgment** propose donc 3 méthodes :

- **void acknowledge()** : Utilisé pour le commit manuel en mode individuel ou BATCH
- **default void nack(int index, Duration sleep)** : Acknowledge négatif d'un index d'un batch  
=> Commite les offsets avant l'index puis repositionne l'offset à index pour reconsumer les messages
- **default void nack(Duration sleep)** : Acknowledge négatif du record courant  
=> L'offset est repositionner sur l'enregistrement



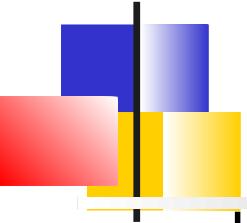
# Configuration sans SpringBoot

---

```
@Bean
KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>>
    kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
        new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());
    factory.setConcurrency(3);
    factory.getContainerProperties().setPollTimeout(3000);
    return factory;
}

@Bean
public ConsumerFactory<Integer, String> consumerFactory() {
    return new DefaultKafkaConsumerFactory<>(consumerConfigs());
}

@Bean
public Map<String, Object> consumerConfigs() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, embeddedKafka.getBrokersAsString());
    ...
    return props;
}
```



# Configuration SpringBoot

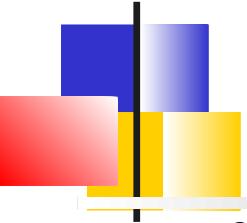
---

Les valeurs de configuration relatives à *DefaultConsumerFactory* sont fixées par la propriété ***spring.kafka.consumer*** :

- *bootstrap-servers, group-id, Deserialisers, enable-auto-commit, auto-offset-reset, ...*

Les valeurs relatives influant le container et l'implémentation de MessageListener sont fixées par ***spring.kafka.listener*** :

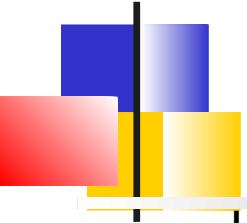
- *type : batch | single*
- *concurrency, client-id, log-container-config*
- *ack-mode, ack-count, ack-time*
- *poll-timeout, idle-between-poll*
- *monitor-poll*



# Exemple

---

```
spring:  
  kafka:  
    consumer:  
      group-id: consumer  
      bootstrap-servers:  
        - localhost:9092  
        - localhost:9192  
      auto-offset-reset: earliest  
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer  
      value-deserializer: org.formation.model.JsonDeserializer  
      enable-auto-commit: true  
  
    listener:  
      concurrency: 3 # Container concurrent  
      log-container-config: true
```



# @KafkaListener

L'annotation **@KafkaListener** permet de désigner une méthode comme *MessageListener*.

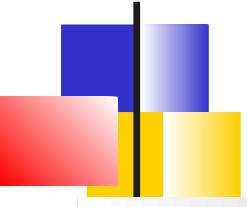
Ses attributs<sup>1</sup> surchargent ceux de *application.yml* dans le cas de SpringBoot permettent de spécifier :

- Le(s) topics à écouter ou même plus précisément les partitions et les offsets
- L'identité du groupe de consommateur
- Le préfixe des clientId
- La propriété *concurrency*
- N'importe quelle propriété

La méthode supporte différents arguments :

- Données ou *ConsumerRecord*
- *Acknowledgment* : Commits manuels
- Entêtes du message, l'argument doit être annoté

1. Supporte les expressions SpEL "#{someBean.someProperty}" ou les propriétés "\${some.property}"



# Exemples

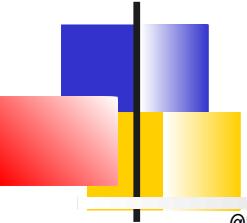
---

```
@KafkaListener(id = "myListener", topics = "myTopic", concurrency = "${listen.concurrency:3}")
public void listen(String data) {

@KafkaListener(id = "thing2", topicPartitions =
    { @TopicPartition(topic = "topic1", partitions = { "0", "1" }),
      @TopicPartition(topic = "topic2", partitions = "0",
                      partitionOffsets = @PartitionOffset(partition = "1", initialOffset = "100")) })
public void listen(ConsumerRecord<?, ?> record) {

@KafkaListener(id = "cat", topics = "myTopic", containerFactory = "kafkaManualAckListenerContainerFactory")
public void listen(String data, Acknowledgment ack) {
    ...
    ack.acknowledge();
}

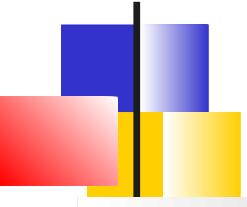
@KafkaListener(id = "qux", topicPattern = "myTopic1")
public void listen(@Payload @Valid MyObject foo,
    @Header(name = KafkaHeaders.RECEIVED_KEY, required = false) Integer key,
    @Header(KafkaHeaders.RECEIVED_PARTITION) int partition,
    @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
    @Header(KafkaHeaders.RECEIVED_TIMESTAMP) long ts
) {
```



# Exemples en mode batch

---

```
@KafkaListener(id = "list", topics = "myTopic", containerFactory = "batchFactory")
public void listen(List<String> list) {  
  
@KafkaListener(id = "listCRs", topics = "myTopic", containerFactory = "batchFactory")
public void listen(List<ConsumerRecord<Integer, String> list) {  
  
@KafkaListener(id = "listCRsAck", topics = "myTopic", containerFactory = "batchFactory")
public void listen(List<ConsumerRecord<Integer, String>> list, Acknowledgment ack) {  
  
@KafkaListener(id = "list", topics = "myTopic", containerFactory = "batchFactory")
public void listen(List<String> list,
                  @Header(KafkaHeaders.RECEIVED_KEY) List<Integer> keys,
                  @Header(KafkaHeaders.RECEIVED_PARTITION) List<Integer> partitions,
                  @Header(KafkaHeaders.RECEIVED_TOPIC) List<String> topics,
                  @Header(KafkaHeaders.OFFSET) List<Long> offsets) {  
  
@KafkaListener(id = "listMsgAck", topics = "myTopic", containerFactory = "batchFactory")
public void listen15(List<Message<?>> list, Acknowledgment ack) {  
  
@KafkaListener(id = "listMsgAckConsumer", topics = "myTopic", containerFactory = "batchFactory")
public void listen16(List<Message<?>> list, Acknowledgment ack, Consumer<?, ?> consumer) {
    ...
```



# Annotation de classe

---

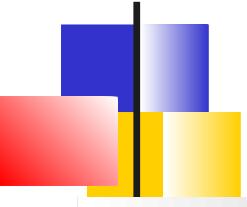
`@KafkaListener` peut également être utilisée sur la classe permettant de mutualiser des propriétés du listener.

Ensuite des annotations `@KafkaHandler` précisent les méthodes de réception. La méthode est sélectionnée en fonction de la conversion de la charge utile.

```
@KafkaListener(id = "multi", topics = "myTopic")
static class MultiListenerBean {
    @KafkaHandler
    public void listen(String foo) { ... }

    @KafkaHandler
    public void listen(Integer bar) { ... }

    @KafkaHandler(isDefault = true)
    public void listenDefault(Object object) { ... }
}
```

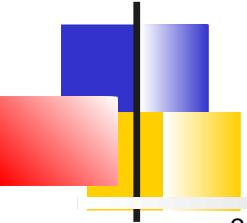


# Rééquilibrage des listeners

*ContainerProperties* a une propriété ***consumerRebalanceListener***, qui prend une implémentation de *ConsumerRebalanceListener* (API Kafka)

- Si pas renseignée, un logger par défaut est associé. Il trace en mode INFO les rééquilibrages
- Sinon Spring fournit une sous-interface ***ConsumerAwareRebalanceListener*** qui peut être utilisée

```
public interface ConsumerAwareRebalanceListener extends ConsumerRebalanceListener {  
  
    void onPartitionsRevokedBeforeCommit(Consumer<?, ?> consumer, Collection<TopicPartition> partitions);  
  
    void onPartitionsRevokedAfterCommit(Consumer<?, ?> consumer, Collection<TopicPartition> partitions);  
  
    void onPartitionsAssigned(Consumer<?, ?> consumer, Collection<TopicPartition> partitions);  
  
    void onPartitionsLost(Consumer<?, ?> consumer, Collection<TopicPartition> partitions);  
  
}
```



# Exemple :

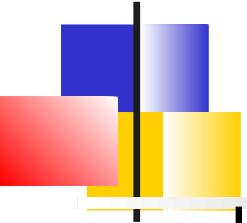
## Stockage des offsets dans support externe

```
containerProperties.setConsumerRebalanceListener(new ConsumerAwareRebalanceListener() {

    @Override
    public void onPartitionsRevokedBeforeCommit(Consumer<?, ?> consumer,
                                                Collection<TopicPartition> partitions) {
        // acknowledge any pending Acknowledgments (if using manual acks)
    }

    @Override
    public void onPartitionsRevokedAfterCommit(Consumer<?, ?> consumer, Collection<TopicPartition> partitions)
    {
        // ...
        store(consumer.position(partition));
        // ...
    }

    @Override
    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
        // ...
        consumer.seek(partition, offsetTracker.getOffset() + 1);
        // ...
    }
});
```

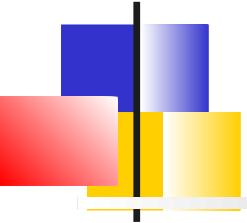


# Annotation `@SendTo`

---

L'annotation `@SendTo` permet de renvoyer la valeur de retour vers un topic :

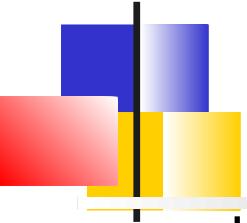
- `@SendTo("someTopic")`
- `@SendTo("#{someExpression}")`
- `@SendTo("!{someExpression}")` route vers le sujet déterminé en évaluant l'expression au moment de l'exécution. L'objet `#root` pour l'évaluation a trois propriétés :
  - **request** : l'objet `ConsumerRecord` entrant (ou l'objet `ConsumerRecords` pour un écouteur par lots).
  - **source** : l'objet `Message<?>` converti à partir de la requête.
  - `result` : le résultat renvoyé par la méthode.
- `@SendTo` : équivalent à  
`!{source.headers['kafka_replyTopic']}`



# Spring-Kafka

---

Introduction  
Production de message  
Consommation  
**Transaction**  
Sérialisation / Désérialisation  
Traitement des exceptions



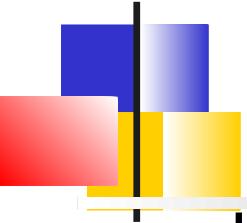
# Support Spring-kafka

---

Le support Spring pour les transactions :

- **KafkaTransactionManager**: Implémentation de *PlatformTransactionManager* qui permet le support classique Spring : `@Transactional`, `TransactionTemplate` etc
- Transactions locales avec *KafkaTemplate*, transaction indépendante Kafka
- Synchronisation avec d'autres gestionnaires de transaction (*TransactionManager* BD par exemple)
- **KafkaMessageListenerContainer transactionnel** : Permettant la sémantique Exactly Once

Les transactions sont activées dès que la propriété ***transactionIdPrefix*** est renseignée



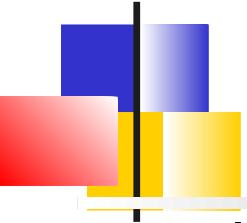
# Producteurs transactionnels

---

Lorsque les transactions sont activées, au lieu de gérer un seul producteur partagé, Spring maintient un cache de producteurs transactionnels.

La propriété Kafka *transactional.id* de chaque producteur est ***transactionIdPrefix + n***, où n commence par 0 et est incrémenté pour chaque nouveau producteur.

*transactionIdPrefix* doit être unique par instance de processus/application.



# Transaction locale Kafka

---

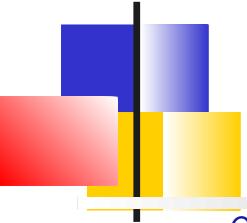
L'ensemble des messages envoyés durant la transaction sont validés ou aucun.

Annotation :

```
@Transactional("kafkaTransactionalManager")
public void process(List<Thing> things) {
    things.forEach(thing -> this.kafkaTemplate.send("topic", thing));
}
```

*KafkaTemplate et **executeInTransaction***

```
boolean result = template.executeInTransaction(t -> {
    t.sendDefault("thing1", "thing2");
    t.sendDefault("cat", "hat");
    return true;
});
```



# Envoi transactionnel synchronisé

---

@Transactional

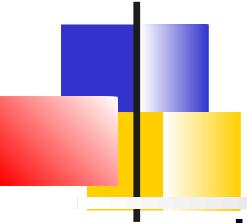
```
public void process(List<Thing> things) {  
    things.forEach(thing -> this.kafkaTemplate.send("topic", thing));  
    updateDb(things);  
}
```

A l'entrée de méthode, l'intercepteur de *@Transactional* démarre la transaction synchronisée avec le gestionnaire de transaction (ici la BD)

A la sortie de méthode sans exception, la transaction BD est committée, suivie de la transaction Kafka.

Attention : Si le 2nd commit échoue, une exception sera envoyé à l'appelant qui devra compenser les effets de la 1ère transaction.

Voir Transactional Outbox Pattern



# Container transactionnel et sémantique Exactly-Once

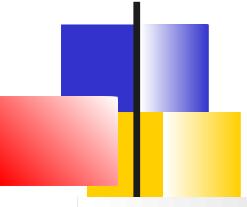
---

L'utilisation de transactions permet la sémantique *Exactly Once* :

Pour une séquence ***read→process→write***, la séquence est terminée exactement une fois

Cette sémantique est donc valable pour les consommateurs de messages qui écrivent dans un topic Kafka.

Cela est permis grâce à l'API Kafka  
*producer.sendOffsetsToTransaction()*.

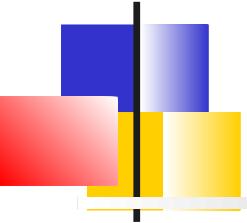


# Mécanisme

---

Lorsque le listener est configuré pour s'exécuter dans une transaction, toutes les opérations de KafkaTemplate font partie de la transaction.

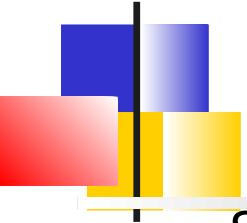
- Si le listener traite avec succès l'enregistrement (ou le lot), le conteneur envoie le ou les offsets avec la transaction à l'aide de `producer.sendOffsetsToTransaction()`, puis commit la transaction.
- Si l'écouteur génère une exception, la transaction est annulée et le consommateur est repositionné afin que le ou les enregistrements annulés puissent être récupérés lors du prochain poll



# Spring-Kafka

---

Introduction  
Production de message  
Consommation  
Transaction  
**Sérialisation / Désérialisation**  
Traitement des exceptions



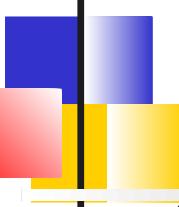
# Introduction

---

Spring Kafka fournit des sérialiseurs/désérialiseurs pouvant être configurés dans les producteurs/consommateurs :

- **Format String** : `StringSerializer/ParseStringSerializer`
- **Format JSON** : `JsonSerializer/JsonDeserializer`
- **Delegating(De)Serializer** : Permettant d'utiliser un (de)serializer différent en fonction d'une clé, d'un type ou d'un topic,
- **RetryingDeserializer** : Permettant plusieurs essais de désérialisation
- **ErrorHandlingDeserializer** : Déserialiseur permettant de traiter les erreurs de désérialisation

Spring Kafka fournit également un intégration avec `SpringMessaging` permettant de convertir les messages Kafka en classe **Message**



# Sérialisation String

---

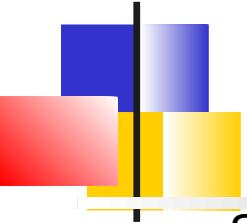
Spring Kafka fournit les classes ***ToSerializer*** et ***ParseStringDeserializer*** s'appuyant respectivement sur les méthodes *toString* ou *Function<String>* ou *BiFunction<String, Headers>*

Exemples :

```
ToSerializer<Thing> thingSerializer = new ToSerializer<>();
ParseStringDeserializer<Thing> deserializer = new ParseStringDeserializer<>(Thing::parse)
```

Par défaut, *ToSerializer* précise l'information de type dans l'entête du message. Cette information peut être utilisée par *ParseStringDeserializer* du côté du consommateur.

```
ParseStringDeserializer<Object> deserializer = new ParseStringDeserializer<>((str, headers) -> {
    byte[] header = headers.lastHeader(ToSerializer.VALUE_TYPE).value();
    String entityType = new String(header);
    if (entityType.contains("Thing")) { return Thing.parse(str); }
    else { // ...parsing logic } })
```

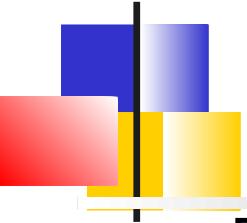


# *JsonSerializer / JsonDeserializer*

---

SpringKafka offre du support pour la sérialisation JSON basé sur **ObjectMapper** de Jackson. Propriétés de configuration :

- *JSONSerializer*
  - **ADD\_TYPE\_INFO\_HEADERS** (défaut true) : Ajoute des informations de type dans les entêtes
  - **TYPE\_MAPPINGS** : Permet de définir des correspondances de type
- *JsonDeserializer*
  - **USE\_TYPE\_INFO\_HEADERS** (défaut true) : Utilise les informations de type pour la désérialisation
  - **KEY\_DEFAULT\_TYPE, VALUE\_DEFAULT\_TYPE** : Types par défaut si l'entête de type n'est pas présente
  - **TRUSTED\_PACKAGES** : Pattern de packages autorisés pour la désérialisation. Par défaut : `java.util, java.lang`
  - **TYPE\_MAPPINGS** : Permet de définir des correspondances de type
  - **KEY\_TYPE\_METHOD, VALUE\_TYPE\_METHOD**: Définition d'une méthode pour déterminer le type à désérialiser



# Sérialisation JSON

## Comportement par défaut

---

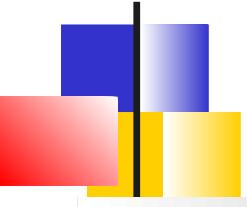
Par défaut, le nom complet de la classe sérialisé est présent dans une entête du message : **TypeId**

Au moment de la désérialisation,

- Spring désérialise dans le type indiqué
- Il vérifie que la classe appartient à un package de confiance

=> Cela implique :

- que la classe sérialisée soit présente dans le même package du côté consommateur et producteur
- Que la propriété ***spring.json.trusted.packages*** spécifie le package de la classe



# Mapping de types

---

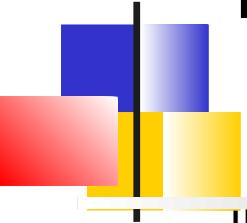
Le mapping de type permet aux consommateurs/producteurs de ne pas partager la même structure de package :

- Côté producteur, le nom de classe est mappé sur un jeton.
- Côté consommateur, le jeton dans l'en-tête de type est mappé sur une classe.

```
senderProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class);
senderProps.put(JsonSerializer.TYPE_MAPPINGS, "cat:com.mycat.Cat, hat:com.myhat.hat");
...
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, JsonDeserializer.class);
consumerProps.put(JsonDeserializer.TYPE_MAPPINGS, "cat:com.yourcat.Cat, hat:com.yourhat.hat");
```

SpringBoot :

```
spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.producer.properties.spring.json.type.mapping=cat:com.mycat.Cat,hat:com.myhat.Hat
```



# Méthodes pour déterminer les types

Il est possible de configurer le déserialiseur afin qu'il utilise une méthode pour déterminer le type cible

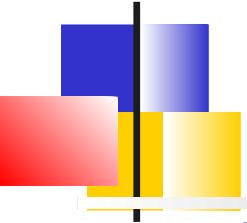
```
spring.kafka.consumer.properties.spring.json.type.value.type.method=org.formation.TypeResolver.myMethod
```

La méthode doit être *public static*, retourner un *Jackson JavaType*. 3 signatures possibles :

- (String topic, byte[] data, Headers headers),
- (byte[] data, Headers headers)
- (byte[] data)

```
JavaType thing1Type = TypeFactory.defaultInstance().constructType(Thing1.class);
JavaType thing2Type = TypeFactory.defaultInstance().constructType(Thing2.class);

public static JavaType thingOneOrThingTwo(byte[] data, Headers headers) {
    return data[21] == '1' ? thing1Type : thing2Type;
}
```

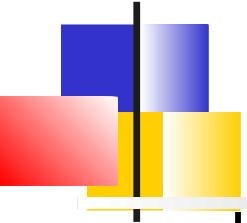


# Délégation

---

***DelegatingSerializer / DelegatingDeserializer*** permettent de sélectionner différents dé/sérialiseur en fonction :

- Des entêtes
- Du type du message produit
- Le nom du topic



# Par entête

---

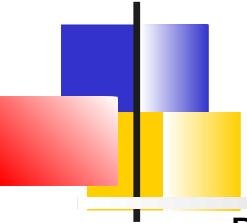
Les producteurs envoient des clés dans les entêtes  
*VALUE\_SERIALIZATION\_SELECTOR* et  
*KEY\_SERIALIZATION\_SELECTOR*

Les consommateurs utilisent les mêmes entêtes pour sélectionner le déserialiseur correspondant à la clé

La configuration consiste donc à fournir des Maps des 2 côtés :

```
producerProps.put(DelegatingSerializer.VALUE_SERIALIZATION_SELECTOR_CONFIG,  
    "thing1:com.example.MyThing1Serializer, thing2:com.example.MyThing2Serializer")
```

```
consumerProps.put(DelegatingDeserializer.VALUE_SERIALIZATION_SELECTOR_CONFIG,  
    "thing1:com.example.MyThing1Deserializer,  
    thing2:com.example.MyThing2Deserializer")
```



# Autres types de délégation

---

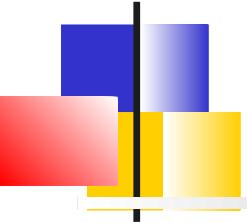
## Par type

@Bean

```
public ProducerFactory<Integer, Object> producerFactory(Map<String, Object> config) {  
    return new DefaultKafkaProducerFactory<>(config,  
        null, new DelegatingByTypeSerializer(Map.of(  
            byte[].class, new ByteArraySerializer(),  
            Bytes.class, new BytesSerializer(),  
            String.class, new StringSerializer())));  
}
```

## Par regexp sur le nom du topic

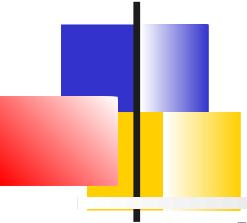
```
producerConfigs.put(DelegatingByTopicSerializer.VALUE_SERIALIZATION_TOPIC_CONFIG,  
    "topic[0-4]:" + ByteArraySerializer.class.getName()  
    + ", topic[5-9]:" + StringSerializer.class.getName());  
...  
ConsumerConfigs.put(DelegatingByTopicDeserializer.VALUE_SERIALIZATION_TOPIC_CONFIG,  
    "topic[0-4]:" + ByteArrayDeserializer.class.getName()  
    + ", topic[5-9]:" + StringDeserializer.class.getName());
```



# Spring-Kafka

---

Introduction  
Production de message  
Consommation  
Transaction  
Sérialisation / Désérialisation  
**Traitements des exceptions**

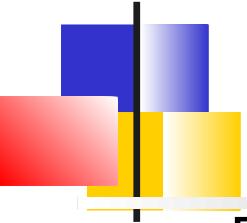


# Types d'erreur

---

Lors de la consommation de messages différents types d'erreur peuvent survenir

- Erreur de désrialisation Kafka : Dans ce cas le message n'est pas récupéré par le framework Spring
- Erreur de traitement du message : Un exception est lancée lors du traitement



# Configuration du traitement d'erreur

---

Par défaut, le message fautif est sauté (skip) et la consommation continue

Il est possible de configurer des gestionnaires d'exception au niveau container ou listener (attribut *errorHandler*)

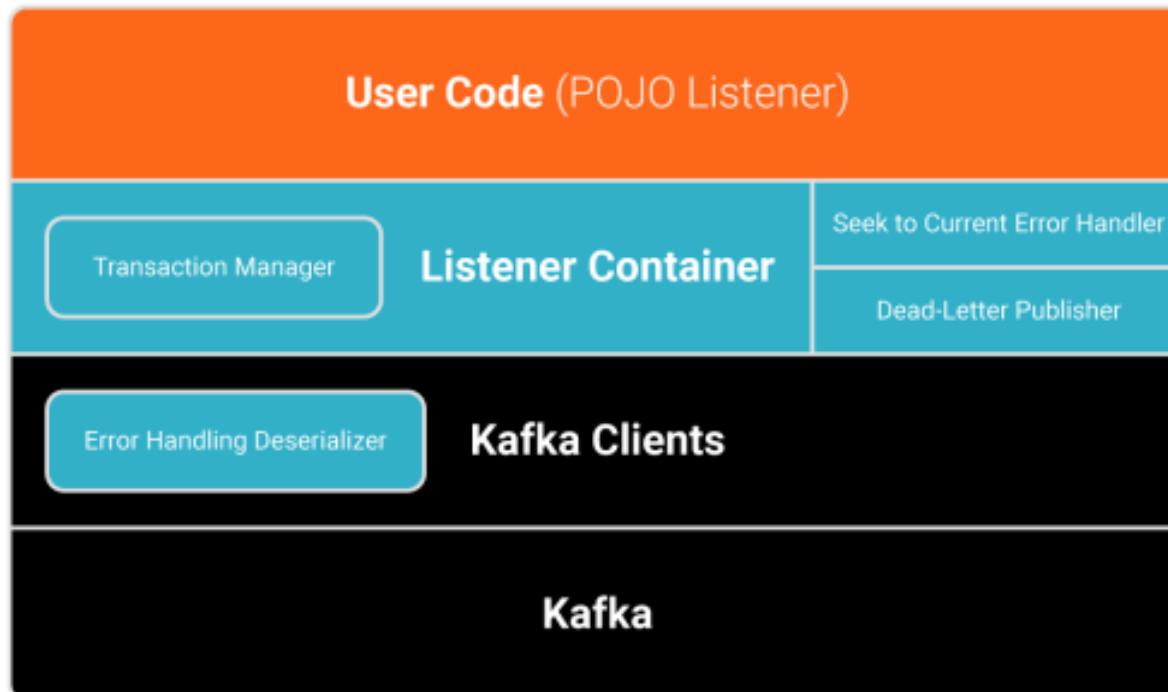
Les gestionnaires peuvent retenter un certain nombre de fois la consommation du message en erreur

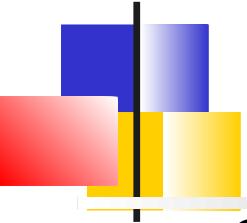
- De façon synchrone : les messages suivants sont bloqués pendant les tentatives
- De façon asynchrone : les messages suivants sont consommés pendant les tentatives. (L'ordre n'est plus respecté)

Il est possible de configurer un déserialiseur Spring gérant les erreurs de sérialisation

# Support Spring

● Spring





# Configuration Listener

---

@KafkaListener propose l'attribut **errorHandler** permettant de définir le nom du bean gérant les erreurs.

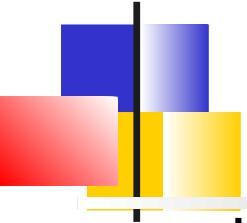
Le bean implémente l'interface fonctionnelle :

```
Object handleError(Message<?> message, ListenerExecutionFailedException exception)
default Object handleError(Message<?> message, ListenerExecutionFailedException exception, Consumer<?, ?
    > consumer)
default Object handleError(Message<?> message, ListenerExecutionFailedException exception, Consumer<?, ?
    > consumer, Acknowledgment ack)
```

Les méthodes donnent accès à l'objet *Message<?>* et à l'exception lancée par le listener encapsulée dans *ListenerExecutionFailedException*.

Le gestionnaire peut lancer l'exception d'origine ou une nouvelle exception vers le container. L'objet renvoyé par le gestionnaire est alors ignoré.

Dans le cas d'un listener annoté avec @SendTo, la valeur retournée par le gestionnaire d'erreur est renvoyé au producteur initial



# Configuration Container

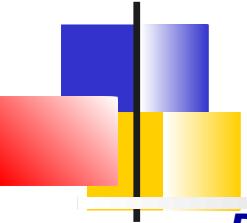
---

L'interface **CommonErrorHandler** permet de configurer un gestionnaire au niveau container qui s'applique pour tous les listeners

- Des implémentations sont fournies

Lorsque on utilise des transactions, aucun gestionnaire n'est configuré par défaut, l'exception annule la transaction.

L'interface AfterRollbackProcessor permet une gestion des erreurs pour les conteneurs transactionnels. Si il envoie une exception, la transaction est annulée



# DefaultErrorHandler

---

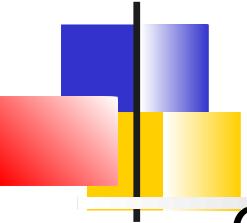
**DefaultErrorHandler** est utilisé pour rejouer un message en erreur. Le traitement est synchrone

**N** tentatives sont essayées (10 par défaut),

- si elles échouent toutes :
  - Soit un simple trace
  - Soit un *DeadLetterPublishingRecoverer* est configuré, permettant d'envoyer le message vers un topic dead Letter (même nom avec le suffixe DLT)
- si une réussit : la consommation continue

Avant de réessayer la consommation, il committe les messages traités. La séquence d'évènements est la suivante :

- 1) Committe les offsets des records avant le message fautif
- 2) Si le nombre de tentatives n'est pas atteint, effectue un seeks afin que les messages non traités incluant le message fautif soient redélivrés
- 3) Si max de retry, tente un recovery du message fautif et se repositionne après le message fautif.
- 4) Si max retry et la recovery lance une exception, on recommande les retry



# Configuration

---

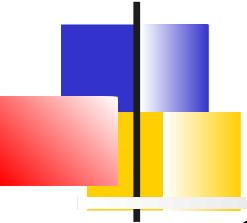
On peut configurer le gestionnaire avec un récupérateur personnalisé (BiConsumer) et un BackOff qui contrôle les tentatives de livraison et les délais entre chacune d'elles.

2 implémentations de Backoff sont fournies :

- **FixedBackOff** : Délai fixe entre les tentatives
- **ExponentialBackOff** : Délai exponentiel entre les tentatives

L'exemple suivant configure la récupération après trois tentatives :

```
@Bean
public CommonErrorHandler errorHandler(KafkaTemplate<Object, Object> template) {
    return new DefaultErrorHandler(
        new DeadLetterPublishingRecoverer(template), new FixedBackOff(1000L, 2));
}
```

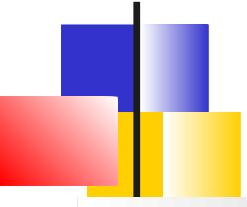


# Exceptions fatales

---

Certaines exceptions sont considérées comme fatales et les tentatives sont ignorées pour ces exceptions ; le récupérateur est invoqué lors du premier échec.

- DeserializationException
- MessageConversionException
- ConversionException
- MethodArgumentResolutionException
- NoSuchMethodException
- ClassCastException

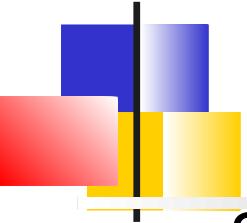


# Listener de la recovery

---

Le gestionnaire d'erreurs peut être configuré avec un ou plusieurs ***RetryListeners***, recevant des notifications de nouvelle tentative et de progression de la récupération.

```
@FunctionalInterface
public interface RetryListener {
    void failedDelivery(ConsumerRecord<?, ?> record, Exception ex, int deliveryAttempt);
    default void recovered(ConsumerRecord<?, ?> record, Exception ex) { }
    default void recoveryFailed(ConsumerRecord<?, ?> record, Exception original, Exception failure) {}
    default void failedDelivery(ConsumerRecords<?, ?> records, Exception ex, int deliveryAttempt) {}
    default void recovered(ConsumerRecords<?, ?> records, Exception ex) { }
    default void recoveryFailed(ConsumerRecords<?, ?> records, Exception original, Exception failure) {}
}
```

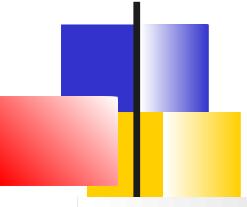


# Autres ErrorHandler

---

Spring fournit d'autres implémentations de CommonErrorHandler :

- **CommonContainerStoppingErrorHandler** : Arrête le container de consommation lors d'une erreur
- **CommonDelegatingErrorHandler** : Délègue à différents gestionnaires en fonction du type de l'exception
- **CommonLoggingErrorHandler** : Log un message d'erreur et continue la consommation
- **CommonMixedErrorHandler** : Permet de définir des gestionnaires différents pour les listener batch et les listener individuel



# Cas des transactions

---

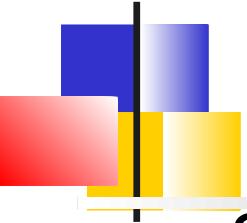
Si le listener (ou le gestionnaire d'erreur si il est présent) génère une exception la transaction est annulée.

Par défaut, tous les enregistrements non traités (y compris l'enregistrement ayant échoué) sont redélivrés, si le listener est en mode batch tout le lot est retraitée

Pour modifier ce comportement, on peut configurer un ***AfterRollbackProcessor***.

Par exemple, abandonner après un certain nombre de tentatives, et publier dans un DLT.

```
@Bean
public DefaultAfterRollbackProcessor errorHandler(BiConsumer<ConsumerRecord<, ?>, Exception> recoverer) {
    return new DefaultAfterRollbackProcessor((record, exception) -> { },
        new FixedBackOff(0L, 2L));
}
```



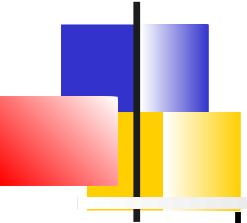
# DLT

---

Spring fournit  
***DeadLetterPublishingRecoverer*** qui publie le message en erreur vers un autre topic (par défaut <topic>-dlt).

Le topic et la partition peuvent être personnalisé :

```
DeadLetterPublishingRecoverer recoverer = new DeadLetterPublishingRecoverer(template,
    (r, e) -> {
        if (e instanceof FooException) {
            return new TopicPartition(r.topic() + ".Foo.failures", r.partition());
        }
        else {
            return new TopicPartition(r.topic() + ".other.failures", r.partition());
        }
    });
CommonErrorHandler errorHandler = new DefaultErrorHandler(recoverer, new FixedBackOff(0L, 2L));
```

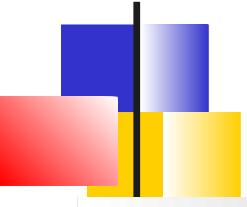


# Entêtes

---

Les informations sur l'erreur sont renseignées dans les entêtes du Record :

- DLT\_EXCEPTION\_FQCN: La classe de l'exception
- DLT\_EXCEPTION\_CAUSE\_FQCN: L'exception cause
- DLT\_EXCEPTION\_STACKTRACE: La stack trace.
- DLT\_ORIGINAL\_TOPIC: Le topic d'origine.
- DLT\_ORIGINAL\_PARTITION: La partition d'origine
- DLT\_ORIGINAL\_OFFSET: L'offset d'origine
- DLT\_ORIGINAL\_TIMESTAMP: Le timestamp d'origine
- DLT\_ORIGINAL\_CONSUMER\_GROUP: Le groupe de consommateur
- ....



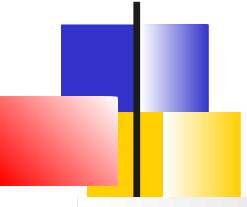
# Non blocking-retry

---

Spring via **@RetryableTopic** et **RetryTopicConfiguration**, permet un traitement asynchrone des messages en erreur

- Le message fautif est transmis à un autre topic avec un délai d'activation.
- A l'échéance du délai, la consommation du topic est tenté, si elle échoue le message est retransmis à un autre topic
- Etc, jusqu'à finir dans un DeadLetterTopic (dlt)

Par exemple, si le topic principal est *myTopic* et que l'on configure un *RetryTopic* avec un BackOff de 1000 ms avec un multiplicateur de 2 et 4 tentatives max  
=> Spring créera les topics *myTopic-retry-1000*, *myTopic-retry-2000*, *myTopic-retry-4000* et *myTopic-dlt*

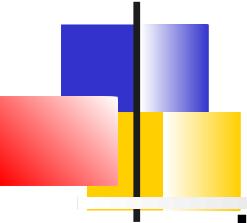


# Erreur de désérialisation

---

Pour contrer les erreurs de désérialisation, 2 alternatives sont fournies par Spring :

- ***RetryingDeserializer*** réessaye la désérialisation lorsque le désérialiseur délégué subit des erreurs transitoires, telles que des problèmes de réseau, pendant la désérialisation.
- ***ErrorHandlingDeserializer*** retourne une valeur null pour la clé ou la valeur et positionne l'exception dans une entête



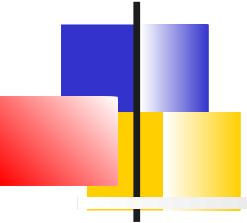
# ErrorHandlingDeserializer

---

Lors d'un listener au niveau Record ou Message, l'entête est testée et si une erreur est présente, l'enregistrement est skipé ou le *ErrorHandler* est appelé

- Il est également possible de configurer une fonction qui génère la valeur de fall-back

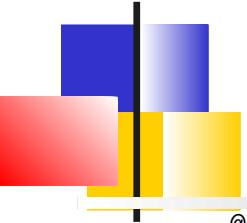
Dans le mode batch, c'est à nous de tester l'entête



# Exemple Configuration

---

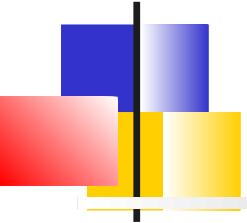
```
... // other props
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, ErrorHandlingDeserializer.class);
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, ErrorHandlingDeserializer.class);
props.put(ErrorHandlingDeserializer.KEY_DESERIALIZER_CLASS, JsonDeserializer.class);
props.put(JsonDeserializer.KEY_DEFAULT_TYPE, "com.example.MyKey")
props.put(ErrorHandlingDeserializer.VALUE_DESERIALIZER_CLASS,
        JsonDeserializer.class.getName());
props.put(JsonDeserializer.VALUE_DEFAULT_TYPE, "com.example.MyValue")
props.put(JsonDeserializer.TRUSTED_PACKAGES, "com.example")
return new DefaultKafkaConsumerFactory<>(props);
```



# Exemple consommation Batch

---

```
@KafkaListener(id = "test", topics = "test")
void listen(List<Thing> in,
    @Header(KafkaHeaders.BATCH_CONVERTED_HEADERS) List<Map<String, Object>> headers) {
    for (int i = 0; i < in.size(); i++) {
        Thing thing = in.get(i);
        if (thing == null
            && headers.get(i).get(SerializationUtils.VALUE_DESERIALIZER_EXCEPTION_HEADER) != null) {
            DeserializationException deserEx =
                ListenerUtils.byteArrayToDeserializationException(this.logger,
                    (byte[]) headers.get(i).get(SerializationUtils.VALUE_DESERIALIZER_EXCEPTION_HEADER));
            if (deserEx != null) {
                logger.error(deserEx, "Record at index " + i + " could not be serialized");
            }
            throw new BatchListenerFailedException("Deserialization", deserEx, i);
        }
        process(thing);
    }
}
```



# Compléments

---

## **Test**

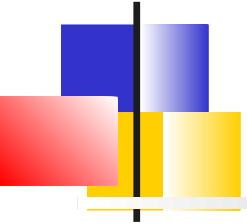
Configuration des listeners

SSL / TLS

Authentification via SASL

ACLs

Quotas

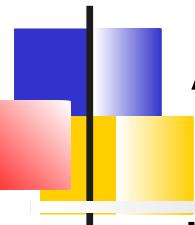


# Support pour le test

---

***spring-kafka-test*** contient quelques utilitaires utiles pour vous aider à tester vos applications.

- Serveur Kafka embarqué (Zookeeper ou mode Kraft)
- Méthodes statiques de KafkaTestUtils
- Des matchers Hamcrest ou Condition Assert4J
- Des Factory pour des MockProducer et des MockConsumer



# Annotation @EmbeddedKafka

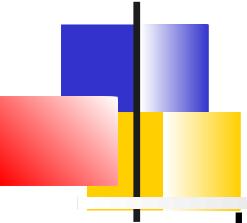
L'annotation **@EmbeddedKafka** peut s'utiliser avec JUnit4 ou Junit5.

- Elle remplace la rule Junit EmbeddedKafkaRule

L'annotation permet de disposer d'un broker embarqué avec des topics pré-créé durant les tests d'une classe de test

Combiner avec `SpringBootTest`, le serveur embarqué est positionné dans le contexte et peut être injecté

Pour éviter des interférences entre 2 classes de test, il est recommandé de ne pas cacher le contexte Spring avec `@DirtiesContext`

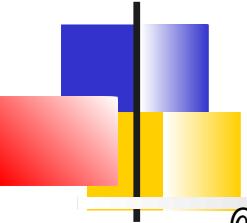


# Principaux attributs

---

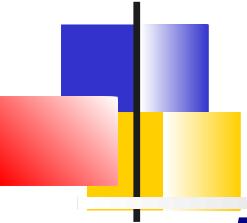
L'annotation `@EmbedeedKafka` permet de définir les attributs suivants :

- **`count`** : Le nombre de brokers
- **`topics`** : Les topics qui seront précréés
- **`partition`** : Le nombre de partitions des topics
- **`kraft`** : Mode kraft ou non
- `BrokerProperties` : Liste des propriétés à ajouter à la config des brokers
- **`bootstrapServerProperties`** : L'adresse du bootstrap-server (automatiquement correctement configurée dans SpringBoot)
- ...



# Exemple

```
@EmbeddedKafka(count = 3, partitions = 1, topics = { "test-topic" })  
@DirtiesContext  
@SpringBootTest  
class MultiBrokerTest {  
    @Autowired  
    private EmbeddedKafkaBroker embeddedKafkaBroker;  
  
    @Test  
    void testClusterSetup() {  
        List<String> brokers =  
embeddedKafkaBroker.getBrokersAsString();  
        System.out.println("Liste des brokers : " + brokers);  
    }  
}
```

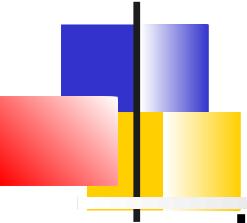


# KafkaTestUtil

---

**KafkaTestUtil** offre certaines méthodes statiques :

- Permettant de configurer les clients avec le serveur embarqué
  - *consumerProps*
  - *producerProps*
- Consommer des messages
  - *getOneRecord*
  - *getSingleRecord*
  - *getRecords*
- Interroger les offsets :
  - *getCurrentOffset*
  - *getEndOffsets*

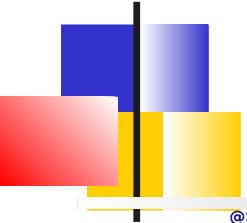


# Test d'un producteur

---

Le test de la production peut consister en :

- Démarrer un serveur embarqué
- S'assurer que dans la configuration de test le producteur utilise le serveur embarqué
- Initialiser un consommateur lié à KafkaTestUtil
- Déclencher la méthode qui produit un message
- Vérifier avec KafkaTestUtil



# Exemple

```
@SpringBootTest(properties = "spring.kafka.bootstrap-servers=localhost:9092")
@EmbeddedKafka(partitions = 1, topics = "{$app.coursier-channel}", brokerProperties = { "listeners=PLAINTEXT://localhost:9092", "port=9092" })
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class PositionControllerTest {

    @Autowired
    private EmbeddedKafkaBroker embeddedKafkaBroker;

    private Consumer<Long, String> consumer;

    @BeforeAll
    void setUp() {
        Map<String, Object> consumerProps = KafkaTestUtils.consumerProps("test-topic", "true", embeddedKafkaBroker);
        consumerProps.put("auto.offset.reset", "earliest");

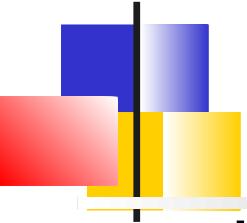
        consumer = new DefaultKafkaConsumerFactory<>(consumerProps, new LongDeserializer(), new StringDeserializer())
            .createConsumer();
        consumer.subscribe(Collections.singleton("test-topic"));
    }

    @AfterAll
    void tearDown() {
        consumer.close();
    }

    @Test
    public void testSend() throws ExecutionException, InterruptedException {

        // Déclenchement de l'envoi
        ConsumerRecord<Long, String> record = KafkaTestUtils.getSingleRecord(consumer, coursierChannel, Duration.ofSeconds(10));

        // Vérifications du contenu du message
    }
}
```

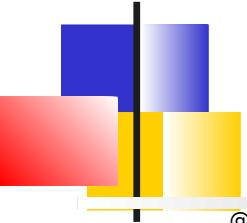


# Test d'un consommateur

---

Le test d'un consommateur peut consister en :

- Démarrer un serveur embarqué
- S'assurer que les affectations de partitions sont faites et que l'on peut démarrer la consommation
- Instancier un KafkaTemplate lié au serveur embarqué
- Envoyer des messages de test
- Vérifier que la méthode de consommation s'est bien passée



# Exemple

---

```
@SpringBootTest(properties = "spring.kafka.consumer.auto-offset-reset=earliest")
@EmbeddedKafka(partitions = 10, topics = {"${app.coursier-channel}"}, count = 3, kraft = true)
@DirtiesContext
public class EventHandlerTest {

    @Autowired
    EventHandler eventHandler;

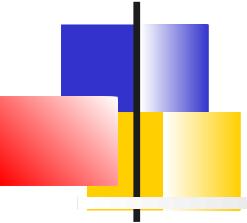
    @BeforeAll
    void setUp() {
        Map<String, Object> producerProps = KafkaTestUtils.producerProps(embeddedKafkaBroker);
        producerFactory = new DefaultKafkaProducerFactory<>(producerProps, new LongSerializer(), new StringSerializer()) ;
    }

    @Test
    public void test() throws ExecutionException, InterruptedException {

        KafkaTemplate kafkaTemplate = new KafkaTemplate(producerFactory);
        ProducerRecord<Long, String> record = new ProducerRecord<Long, String>(coursierChannel, null, 1L, "TEST", headers);
        kafkaTemplate.send(record);

        await().atMost(5, TimeUnit.SECONDS).untilAsserted(() -> {
            //Assertions
        });

    }
}
```



# HamcrestMatcher

---

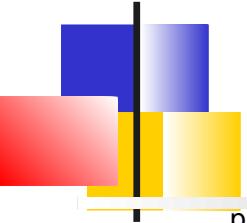
```
public static <K> Matcher<ConsumerRecord<K, ?>> hasKey(K key) { ... }
```

```
public static <V> Matcher<ConsumerRecord<?, V>> hasValue(V value) { ... }
```

```
public static Matcher<ConsumerRecord<?, ?>> hasPartition(int partition) { ... }
```

```
public static Matcher<ConsumerRecord<?, ?>> hasTimestamp(long ts) {  
    return hasTimestamp(TimestampType.CREATE_TIME, ts);  
}
```

```
public static Matcher<ConsumerRecord<?, ?>> hasTimestamp(TimestampType  
    type, long ts) {  
    return new ConsumerRecordTimestampMatcher(type, ts);  
}
```



# AssertJ Condition

```
public static <K> Condition<ConsumerRecord<K, ?>> key(K key) { ... }
```

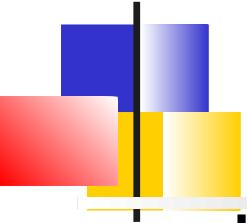
```
public static <V> Condition<ConsumerRecord<?, V>> value(V value) { ... }
```

```
public static <K, V> Condition<ConsumerRecord<K, V>> keyValue(K key, V value) { ... }
```

```
public static Condition<ConsumerRecord<?, ?>> partition(int partition) { ... }
```

```
public static Condition<ConsumerRecord<?, ?>> timestamp(long value) {  
    return new ConsumerRecordTimestampCondition(TimestampType.CREATE_TIME, value);  
}
```

```
public static Condition<ConsumerRecord<?, ?>> timestamp(TimestampType type, long value) {  
    return new ConsumerRecordTimestampCondition(type, value);  
}
```

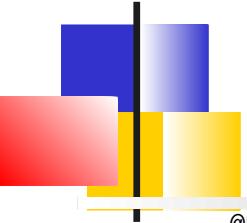


# MockProducer/Consumer

---

La librairie kafka-clients fournit les classes  
**MockConsumer** et **MockProducer**

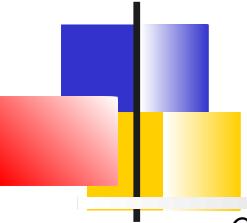
Spring fournit des implémentations  
**MockConsumerFactory** et  
**MockProducerFactory** qui peuvent  
être utilisées à la place des usines par  
défaut pour les KafkaTemplate et les  
containers



# MockConsumerFactory

---

```
@Bean
ConsumerFactory<String, String> consumerFactory() {
    MockConsumer<String, String> consumer = new MockConsumer<>(OffsetResetStrategy.EARLIEST);
    TopicPartition topicPartition0 = new TopicPartition("topic", 0);
    List<TopicPartition> topicPartitions = Collections.singletonList(topicPartition0);
    Map<TopicPartition, Long> beginningOffsets = topicPartitions.stream().collect(Collectors
        .toMap(Function.identity(), tp -> 0L));
    consumer.updateBeginningOffsets(beginningOffsets);
    consumer.schedulePollTask(() -> {
        consumer.addRecord(
            new ConsumerRecord<>("topic", 0, 0L, 0L, TimestampType.NO_TIMESTAMP_TYPE, 0, 0, null, "test1",
                new RecordHeaders(), Optional.empty()));
        consumer.addRecord(
            new ConsumerRecord<>("topic", 0, 1L, 0L, TimestampType.NO_TIMESTAMP_TYPE, 0, 0, null, "test2",
                new RecordHeaders(), Optional.empty()));
    });
    return new MockConsumerFactory(() -> consumer);
}
```



# MockProducer

---

```
@Test
```

```
void givenKeyValue_whenSend_thenVerifyHistory() {
```

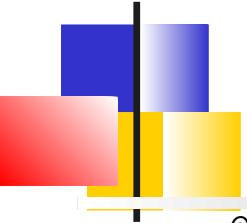
```
    MockProducer mockProducer = new MockProducer<>(true, new  
    StringSerializer(), new StringSerializer());
```

```
    kafkaProducer = new KafkaProducer(mockProducer);
```

```
    Future<RecordMetadata> recordMetadataFuture =  
    kafkaProducer.send("soccer", "{\"site\" : \"baeldung\"}");
```

```
    assertTrue(mockProducer.history().size() == 1);
```

```
}
```

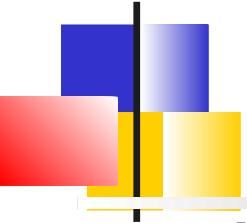


# MockProducerFactory

---

@Bean

```
ProducerFactory<String, String> nonTransFactory() {  
    return new MockProducerFactory<>(() ->  
        new MockProducer<>(true, new StringSerializer(), new  
        StringSerializer()));  
}  
  
@Bean  
ProducerFactory<String, String> transFactory() {  
    MockProducer<String, String> mockProducer =  
        new MockProducer<>(true, new StringSerializer(), new StringSerializer());  
    mockProducer.initTransactions();  
    return new MockProducerFactory<String, String>((tx, id) -> mockProducer,  
        "defaultTxId");  
}
```

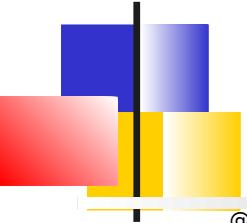


# Alternative TestContainers

---

Une alternative à un serveur embarqué peut être d'utiliser le framework Testcontainers.

Un conteneur Docker est alors démarré lors du test d'intégration.



# Exemple

---

```
@SpringBootTest
@TestPropertySource(
    properties = { "spring.kafka.consumer.auto-offset-reset=earliest" }
)
@Testcontainers
class ProductPriceChangedEventHandlerTest {

    @Container
    static final KafkaContainer kafka = new KafkaContainer(DockerImageName.parse("confluentinc/cp-kafka:7.6.1") );

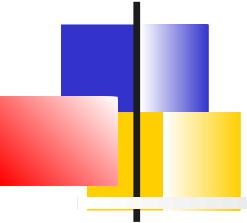
    @DynamicPropertySource
    static void overrideProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.kafka.bootstrap-servers", kafka::getBootstrapServers);
    }

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Test
    void shouldHandleProductPriceChangedEvent() {
        ProductPriceChangedEvent event = new ProductPriceChangedEvent( "P100", new BigDecimal("14.50") );

        kafkaTemplate.send("product-price-changes", event.productCode(), event);

        await()
            .pollInterval(Duration.ofSeconds(3))
            .atMost(10, SECONDS)
            .untilAsserted(() -> {
                // Assertions
            });
    }
}
```



# Compléments

---

Test

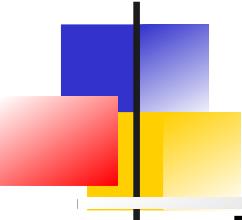
## **Configuration des listeners**

SSL / TLS

Authentification via SASL

ACLs

Quotas



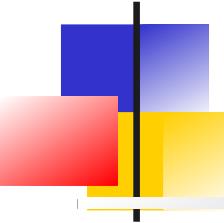
# Introduction

---

Différentes alternatives supportées pour la sécurité :

- Authentification des connections clients vers les brokers via SSL ou SASL
- Authentification des connections entre contrôleurs Kraft OU des brokers vers Zookeeper
- Cryptage des données transférées avec les clients/brokers via TLS/SSL
- Autorisation des opérations read/write/create/delete/... par client
- Possibilité d'intégration avec d'autres mécanismes d'authentification et d'autorisation

Naturellement, dégradation des performances avec SSL

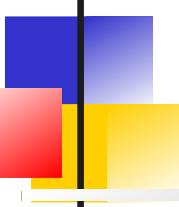


# Listeners

---

Chaque serveur doit définir l'ensemble des *listeners* utilisés pour recevoir les requêtes des clients ainsi que des autres serveurs.

Chaque listener peut être configuré pour authentifier les clients et pour garantir que le trafic entre le serveur et le client est crypté.

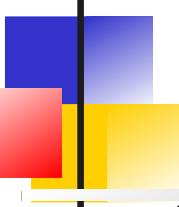


# Combinaisons des protocoles

---

Chaque protocole combine une couche de transport (PLAINTEXT ou SSL) avec une couche d'authentification optionnelle (SSL ou SASL) :

- **PLAIN\_TEXT** : En clair sans authentification. Ne convient qu'à une utilisation au sein de réseaux privés pour le traitement de données non sensibles
- **SSL** : Couche de transport SSL avec authentification client SSL en option. Convient pour une utilisation dans réseaux non sécurisés car l'authentification client et serveur ainsi que le chiffrement sont prise en charge.
- **SASL\_PLAINTEXT** : Couche de transport PLAINTEXT avec authentification client SASL. Ne prend pas en charge le cryptage et convient donc uniquement pour une utilisation dans des réseaux privés.
- **SASL\_SSL** : Couche de transport SSL avec authentification SASL. Convient pour une utilisation dans des réseaux non sécurisés.



# Configuration des listeners

---

Les listeners sont déclarés via la propriété ***listeners*** :

{LISTENER\_NAME}://:{port}

Ou LISTENER\_NAME est un nom descriptif

Exemple :

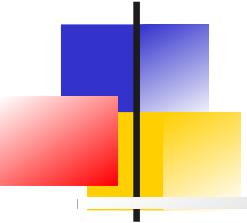
```
listeners=CLIENT://:9092,BROKER://:9095
```

Le protocole utilisé pour chaque listener est spécifié dans la propriété  
***listener.security.protocol.map***

Exemple : `listener.security.protocol.map=CLIENT:SSL,BROKER:PLAINTEXT`

Les protocoles supportés sont : *PLAINTEXT, SSL, SASL\_PLAINTEXT, SASL\_SSL*

Il faut déclarer les listener utilisés pour la communication inter broker via  
***inter.broker.listener.name*** et ***controller.listener.names***



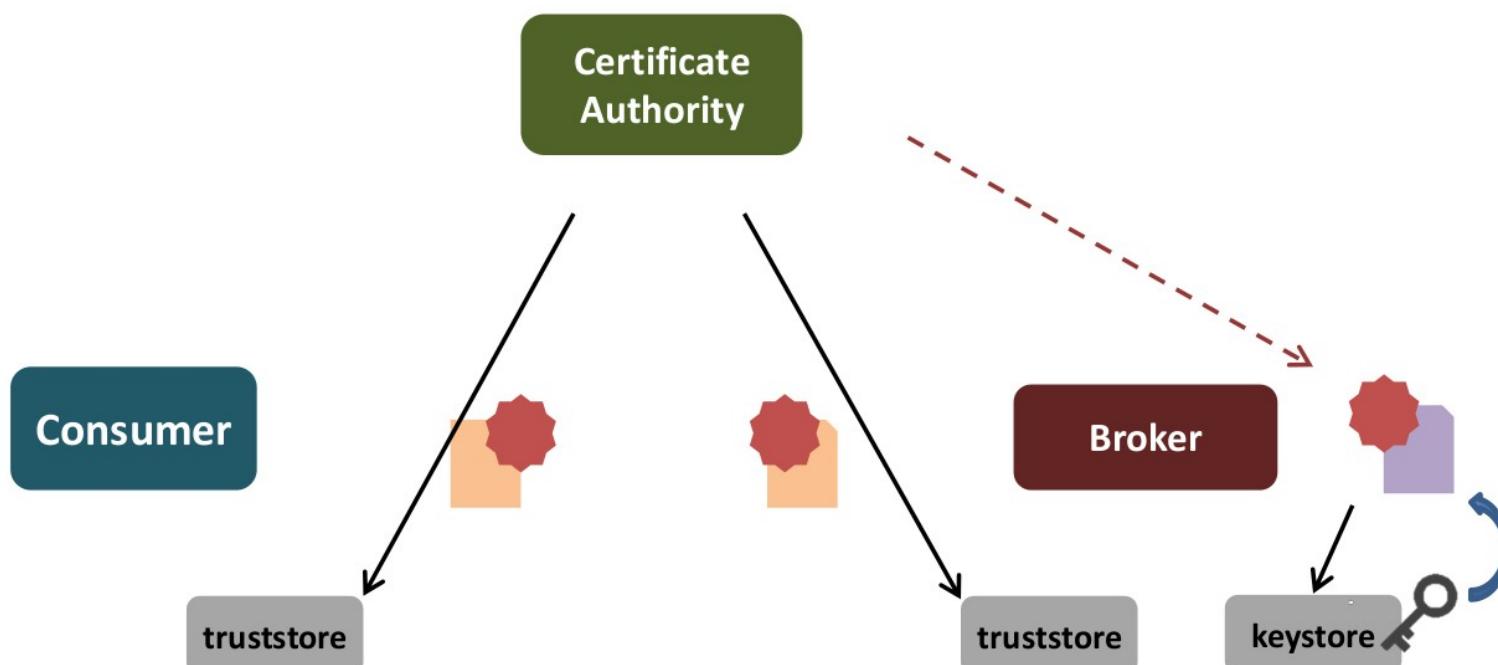
# Compléments

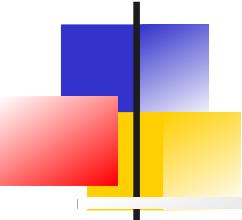
---

Test  
Configuration des listeners  
**SSL/TLS**  
Authentification via SASL  
ACLs  
Quotas

# Certificats

SSL pour le cryptage et l'authentification



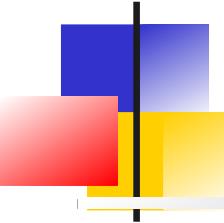


# Configuration TLS et Vérification d'hôte

---

Les certificats des brokers doivent contenir le nom d'hôte du broker en tant que nom alternatif du sujet (SAN) extension ou comme nom commun (CN) pour permettre aux clients de vérifier l'hôte du serveur.

Les certificats génériques utilisant les wildcards peuvent être utilisés pour simplifier l'administration en utilisant le même keystore pour tous les brokers d'un domaine



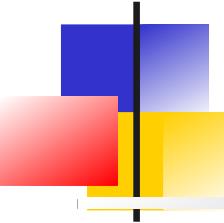
# Configuration du broker

---

*server.properties* :

```
listeners=PLAINTEXT://host.name:port,SSL://  
host.name:port
```

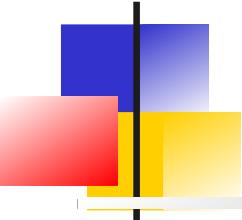
```
ssl.keystore.location=/home/ubuntu/ssl/server.keystore.jks  
ssl.keystore.password=servpass  
ssl.key.password=servpass  
ssl.truststore.location=/home/ubuntu/ssl/server.truststore.jks  
ssl.truststore.password=servpass
```



# Configuration des clients

---

```
security.protocol=SSL  
ssl.truststore.location=/var/private/ssl/  
  client.truststore.jks  
ssl.truststore.password=clipass
```

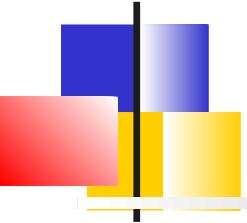


# Authentification des clients via SSL

---

Si l'authentification du client est requise, un *keystore* doit également être créé et la configuration de *client.properties* doit contenir :

```
ssl.keystore.location=/var/private/client.keystore.jks
ssl.keystore.password=test1234
ssl.key.password=test1234
```



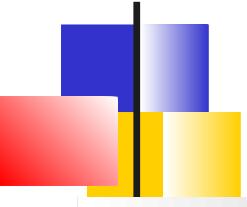
# Compléments

---

Test  
Configuration des listeners  
SSL/TLS

## **Authentification via SASL**

ACLs  
Quotas



# SASL

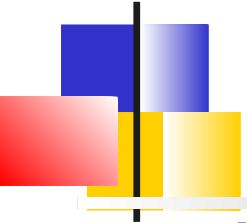
---

*Simple Authentication and Security Layer pour l'authentification*

Kafka utilise JAAS pour la configuration SASL.

Mécanismes:

- GSSAPI: Kerberos
- SCRAM-SHA-256, SCRAM-SHA-512: hashed passwords
- PLAIN: username/password en clair
- OAUTHBEARER (Depuis Kafka 3.x)

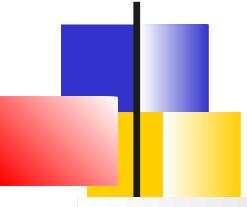


# Configuration

---

L'authentification s'applique :

- Aux brokers (communication inter-broker authentifiée)
- Aux clients (communication client-broker authentifié)
- Dans le mode Kraft, cela concerne également les contrôleurs



# Mécanismes

---

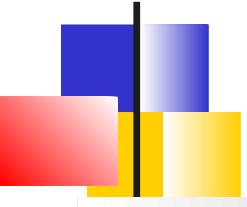
**SASL/Kerberos** : Nécessite un serveur (Active Directory par exemple), d'y créer les *Principals* représentant les brokers. Tous les hosts Kafka doivent être atteignables via leur FQDNs

SpringKafka offre un support pour Keberos

**SASL/PLAIN** est un mécanisme simple d'authentification par login/mot de passe. Doit être utilisé avec TLS.  
Kafka fournit une implémentation par défaut qui peut être étendue pour la production

**SASL/SCRAM (256/512)** (*Salted Challenge Response Authentication Mechanism*) : Mot de passe haché stocké dans Zookeeper

**SASL/OAUTHBEARER** : Basé sur oAuth2 mais pas adapté en l'état à la production



# Configuration JAAS

---

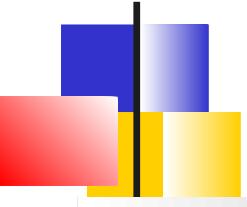
La configuration JAAS s'effectue soit

- via un **fichier jaas** contenant une section *KafkaServer* spécifiant le LoginModule JAAS associé au mécanisme d'authentification.

```
KafkaServer {  
    org.apache.kafka.common.security.scram.ScramLoginModule required  
    username="admin"  
    password="admin-secret";  
};
```

- via la propriété **sasl.jaas.config** préfixée par le mécanisme SASL utilisé

```
listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=  
org.apache.kafka.common.security.scram.ScramLoginModule required \  
username="admin" \  
password="admin-secret";
```



# Exemple Kerberos

---

Utiliser un serveur Kerberos existant ou en installer un<sup>1</sup>

Créer des utilisateurs pour le broker et les clients :

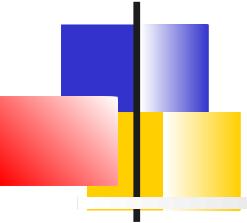
S'assurer que tous les brokers soient accessibles via  
leur FQDNs

Configuration fichier JAAS :

```
KafkaServer {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    storeKey=true  
    keyTab="/etc/security/keytabs/kafka_server.keytab"  
    principal="kafka/kafka1.hostname.com@EXAMPLE.COM";  
};
```

1. <https://help.ubuntu.com/community/Kerberos>,

[https://access.redhat.com/documentation/en-US/Red\\_Hat\\_Enterprise\\_Linux/6/html/Managing\\_Smart\\_Cards/installing-kerberos.html](https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Managing_Smart_Cards/installing-kerberos.html)

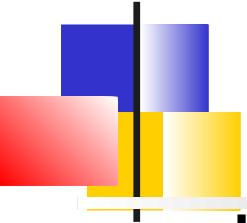


# Exemple Kerberos

---

Configuration *server.properties* :

```
listeners=SASL_PLAINTEXT://host.name:port
security.inter.broker.protocol=SASL_PLAINTEXT
sasl.mechanism.inter.broker.protocol=GSSAPI
sasl.enabled.mechanisms=GSSAPI
# Doit correspondre au nom du principal du broker
sasl.kerberos.service.name=kafka
```



# Spring Kafka et Kerberos

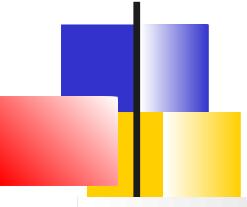
---

Le bean

**KafkaJaasLoginModuleInitializer**  
offre du support pour la configuration  
Kerberos.

```
@Bean
public KafkaJaasLoginModuleInitializer jaasConfig() throws IOException {
    KafkaJaasLoginModuleInitializer jaasConfig = new KafkaJaasLoginModuleInitializer();
    jaasConfig.setControlFlag("REQUIRED");
    Map<String, String> options = new HashMap<>();
    options.put("useKeyTab", "true");
    options.put("storeKey", "true");
    options.put("keyTab", "/etc/security/keytabs/kafka_client.keytab");
    options.put("principal", "kafka-client-1@EXAMPLE.COM");
    jaasConfig.setOptions(options);

    return jaasConfig;
}
```

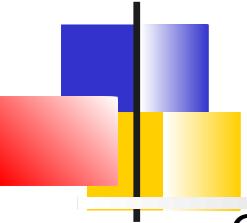


# Exemple SASL PLAIN

---

1. Configurer le fichier JAAS
2. Passer l'emplacement du fichier de configuration JAAS en tant que paramètre JVM à chaque broker
  - Djava.security.auth.login.config=/etc/kafka/kafka\_server\_jaas.conf
3. Configurer SASL dans *server.properties*

```
listeners=SASL_SSL://host.name:port
security.inter.broker.protocol=SASL_SSL
sasl.mechanism.inter.broker.protocol=PLAIN
sasl.enabled.mechanisms=PLAIN
```



# Configuration du client

---

Créer le fichier Jaas

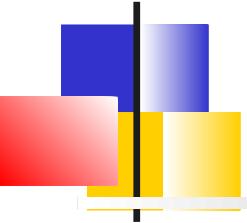
```
KafkaClient {  
    org.apache.kafka.common.security.plain.PlainLoginModule required  
        username="alice"  
        password="alice-secret"  
} ;
```

Ajouter un paramètre JVM

```
-Djava.security.auth.login.config=/home/ubuntu/ssl/client_jaas.conf
```

*client.properties*

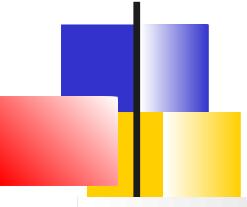
```
security.protocol=SASL_SSL  
sasl.mechanisms=PLAIN
```



# Compléments

---

Test  
Configuration des listeners  
SSL/TLS  
Authentification via SASL  
**ACLs**  
Quotas



# Introduction

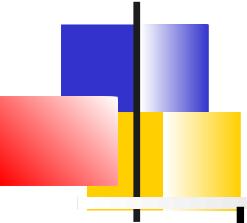
---

Les brokers Kafka gèrent le contrôle d'accès à l'aide d'une API définie par l'interface  
*org.apache.kafka.server.authorizer.Authorizer*

L'implémentation est configurée via la propriété :  
***authorizer.class.name***

Kafka fournit 2 implémentations

- Pour Zookeeper,  
***kafka.security.authorizer.AclAuthorizer***
- En kraft mode  
***org.apache.kafka.metadata.authorizer.StandardAuth orizer***



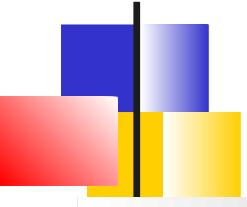
# Autorisation

Les ACLs sont stockées dans les métadonnées gérées par les contrôleurs et peuvent être gérées par l'utilitaire **kafka-acls.sh**

Chaque requête Kafka est autorisée si le *KafkaPrincipal* associé à la connexion a les autorisations pour effectuer l'opération demandée sur les ressources demandées.

Les règles peuvent être exprimées comme suit :

*Principal P is [Allowed/Denied] Operation O From Host H on any Resource R matching ResourcePattern RP*

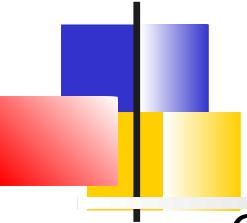


# Exemple

---

*Principals User:Bob et User:Alice sont autorisés à effectuer les opérations Read et Write sur le Topic Test-Topic à partir des IP 198.51.100.0 et IP 198.51.100.1*

```
bin/kafka-acls.sh --bootstrap-servers  
localhost:9092 --add --allow-principal  
User:Bob --allow-principal User:Alice --  
allow-host 198.51.100.0 --allow-host  
198.51.100.1 --operation Read --operation  
Write --topic Test-topic
```



# Ressources et opérations

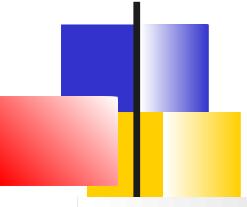
---

Chaque ACL consiste en :

- **Type de ressource** : *Cluster | Topic | Group | TransactionalId*
- **Type de pattern** : *Literal | Prefixed*
- **Nom de la ressource** : Possibilité d'utiliser les wildcards
- **Opération** : *Describe | Create | Delete | Alter | Read | Write | DescribeConfigs | AlterConfigs*
- **Type de permission** : *Allow | Deny*
- **Principal** : De la forme *<principalType>:<principalName>*  
Exemple : *User:Alice, Group:Sales, User :\**
- **Host** : Adresse IP du client, \* si tout le monde

Exemple Complet :

*User:Alice has Allow permission for Write to Prefixed Topic:customer from 192.168.0.1*

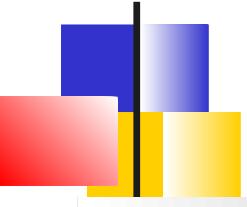


# Règles

---

AclAuthorizer autorise une action s'il n'y a pas d'ACL de DENY qui correspond à l'action et qu'il y a au moins une ACL ALLOW qui correspond à l'action.

- L'autorisation Describe est implicitement accordée si l'autorisation Read, Write, Alter ou Delete est accordée.
- L'autorisation Describe Configs est implicitement accordée si l'autorisation AlterConfigs est accordée.



# Permissions clientes

---

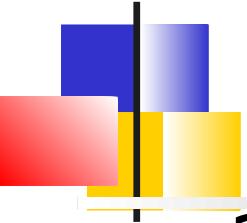
Brokers : **Cluster:ClusterAction** pour autoriser les requêtes de contrôleur et les requêtes de fetch des répliques.

Producteurs simples : **Topic:Write**

- idempotents sans transactions :  
**Cluster:IdempotentWrite**.
- Transactionnels : **TransactionalId:Write** à la transaction et **Group:Read** pour que les groupes de consommateurs valident les offsets.

Consommateurs : **Topic:Read** et **Group:Read** s'ils utilisent la gestion de groupe ou la gestion des offsets.

Clients admin : **Create, Delete, Describe, Alter, DescribeConfigs, AlterConfigs** .



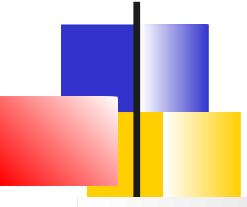
# Exceptions

---

2 options de configuration permettant d'accorder un large accès aux ressources permet de simplifier la mise en place d'ACL à des clusters existants :

- ***super.users*** : Permet de définir les utilisateurs ayant droit à tout
- ***allow.everyone.if.no.acl.found=true*** :

Tous les utilisateurs ont accès aux ressources sans ACL.

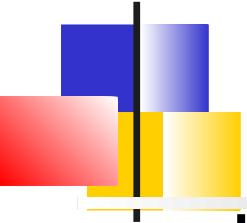


# Exemple

---

*Principals User:Bob et User:Alice sont autorisés à effectuer les opérations Read et Write sur le Topic Test-Topic à partir des IP 198.51.100.0 et IP 198.51.100.1*

```
bin/kafka-acls.sh --authorizer-properties
zookeeper.connect=localhost:2181 --add --
allow-principal User:Bob --allow-principal
User:Alice --allow-host 198.51.100.0 --
allow-host 198.51.100.1 --operation Read
--operation Write --topic Test-topic
```



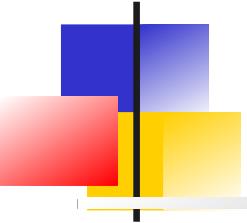
# Audit

---

Les brokers peuvent être configurés pour générer des traces d'audit.

La configuration s'effectue dans *conf/log4j.properties*

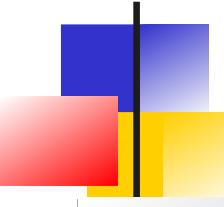
- Le fichier par défaut est *kafka-authorizer.log*
- Le niveau INFO trace les entrées pour chaque refus
- Le niveau DEBUG pour chaque requête acceptée



# Compléments

---

Test  
Configuration des listeners  
SSL/TLS  
Authentification via SASL  
ACLs  
**Quotas**



# Introduction

---

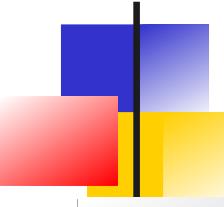
Kafka permet d'appliquer des quotas sur les requêtes pour contrôler les ressources du broker utilisées par les clients.

Deux types de quotas peuvent être appliqués par groupe de client :

- Les quotas de bande passante réseau définissent des seuils de débit
- Les quotas de taux de requête définissent les seuils d'utilisation du processeur en pourcentage du réseau et des threads d'E/S
- Quotas du taux de connexion par IP

L'identité du client correspond au KafkaPrincipal dans un cluster sécurisé ou la propriété applicative client-id.

Tous les clients ayant la même identité partagent leur configuration de quotas



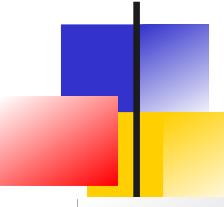
# Quota de bande passante

---

Le seuil de débit d'octets pour chaque groupe de clients.

Chaque groupe peut publier/consommer un maximum de X octets/sec par broker avant d'être limités.

```
kafka-configs.sh --bootstrap-server $BOOT --alter --  
add-config  
'producer_byte_rate=1024,consumer_byte_rate=1024' --  
entity-type users --entity-name <authenticated-  
principal> --command-config /tmp/client.properties
```

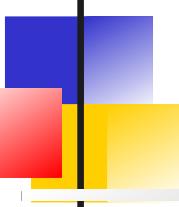


# Taux de requêtes

---

Définis en pourcentage de temps sur les threads d'I/O et de réseau de chaque broker dans une fenêtre temporelle.

```
kafka-configs.sh --bootstrap-server $BOOT --  
alter --add-config 'request_percentage=150'  
--entity-type users --entity-name big-time-  
tv-show-host --command-config  
/tmp/client.properties
```



# Application des quotas

---

Lorsqu'un broker constate une violation de quota, il calcule une estimation du délai nécessaire pour ramener le client sous son quota.

Le broker peut alors :

- Répondre au client avec une demande de délai
- Désactiver le canal de la socket

Selon le client (récent ou non), le client peut soit respecter ce délai, soit l'ignorer.