



Développement d'applications modulaires en Java

Chouki Tibermacine

Chouki.Tibermacine@umontpellier.fr



Plan de l'ECUE

1. (Rappels sur le) Développement d'applications Web avec Java
2. Modulariser les applications Java avec Spring
3. Bien structurer une application Web avec Spring MVC
4. Auto-configurer une application Web avec Spring Boot
5. Sécuriser une application Web avec Spring Security
6. Gérer des données massives avec Apache Kafka et Spring
7. Tester une application Web Spring
8. Écrire des applications Web (API) réactives avec Spring WebFlux

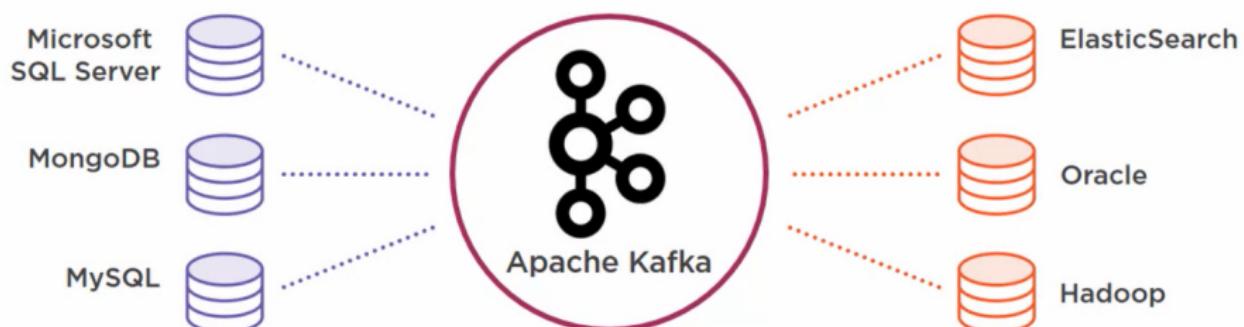
Plan du cours

1. Introduction à Apache Kafka
2. Topics, Partitions et Brokers
3. Produire des messages
4. Consommer des messages
5. Conclusion

Qu'est-ce que “Apache Kafka”

Un système distribué de messagerie publish/subscribe

à grand débit

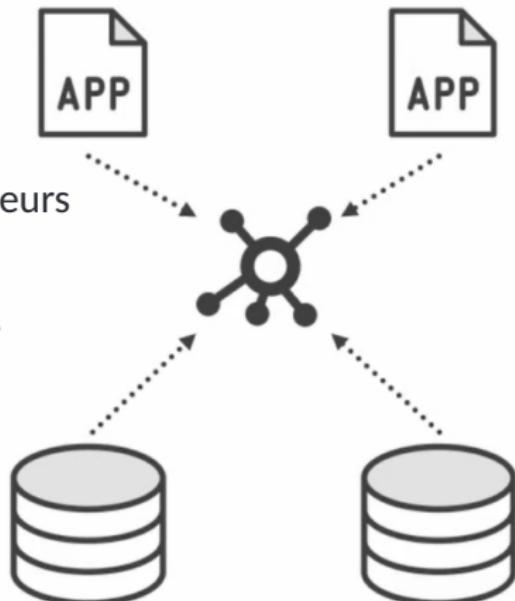


Particularités de Kafka

- Spécialement conçu pour un grand débit de messages (des centaines de millions d'utilisateurs et des milliers de milliards de messages par jour)
- A prouvé son efficacité chez LinkedIn, Netflix, Uber, Airbnb, ...
- Garantit une certaine fiabilité en répliquant les messages
- Traditionnellement, on utilisait la réPLICATION de bases de données, mais c'est peu flexible (la migration d'un fournisseur (*vendor*) à un autre ou une évolution des schémas posent des problèmes, parfois complexes)
- Utilisable pour l'interopérabilité entre des sources de données hétérogènes

Un courtier (*broker*) de messages

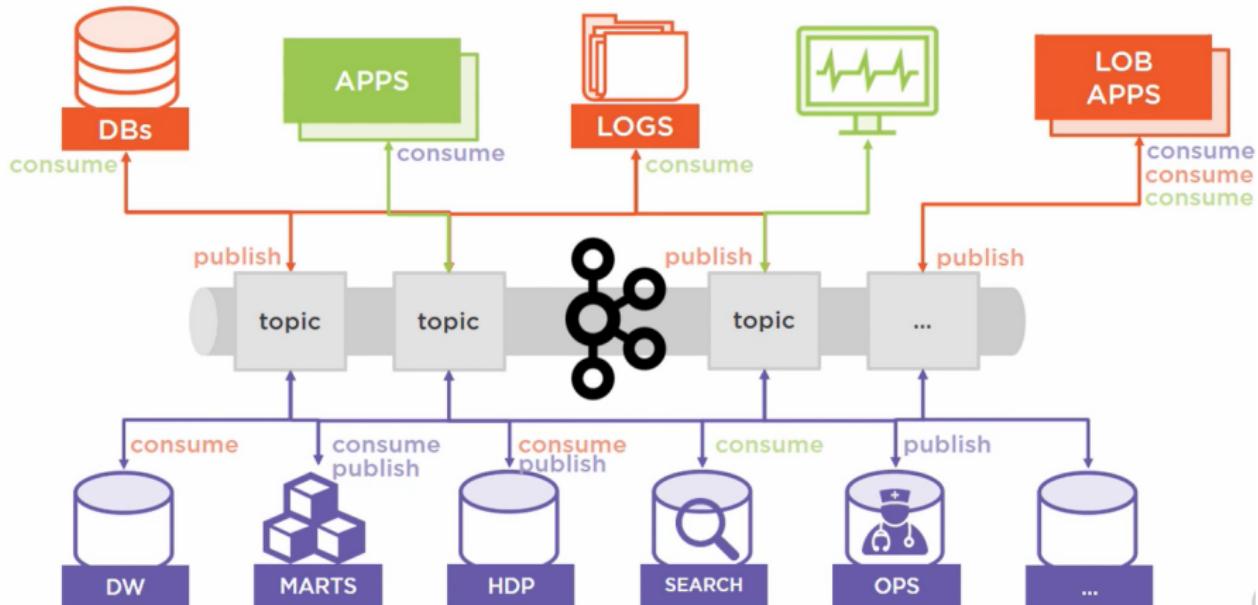
- Permettant de découpler les producteurs et les consommateurs de messages
- Basé sur le pattern publish/subscribe
- Messages durables ou effaçables



LinkedIn avant 2010



LinkedIn après 2010

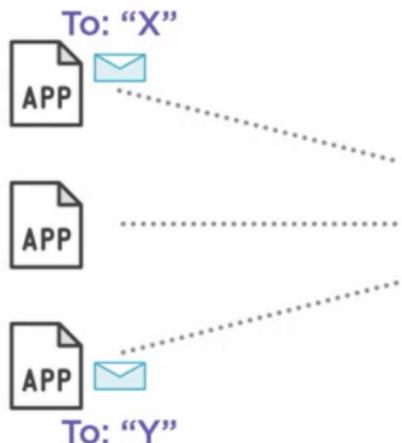


LinkedIn et Kafka

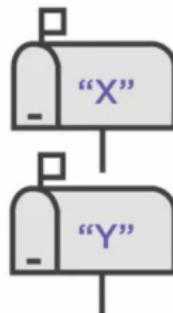
- LinkedIn existe depuis 2003
- Début du développement de Kafka en 2009
- En 2010, c'est opérationnel chez LinkedIn
- En 2011/2012, Kafka est devenu un projet open-source Apache (l'un des plus utilisés de la fondation)
- Aujourd'hui, LinkedIn gère 1.1 k milliards de messages / jour avec Kafka

Producers (publishers), Consumers (subscribers) & Topics

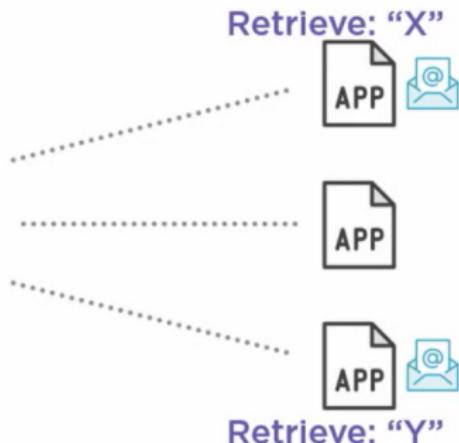
Producers



Topics



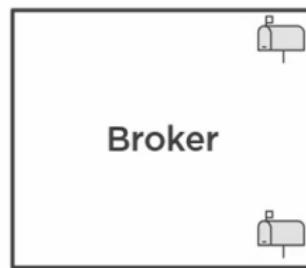
Consumers



Topics = groupes de messages

Courtiers ou *brokers*

Producers



Consumers

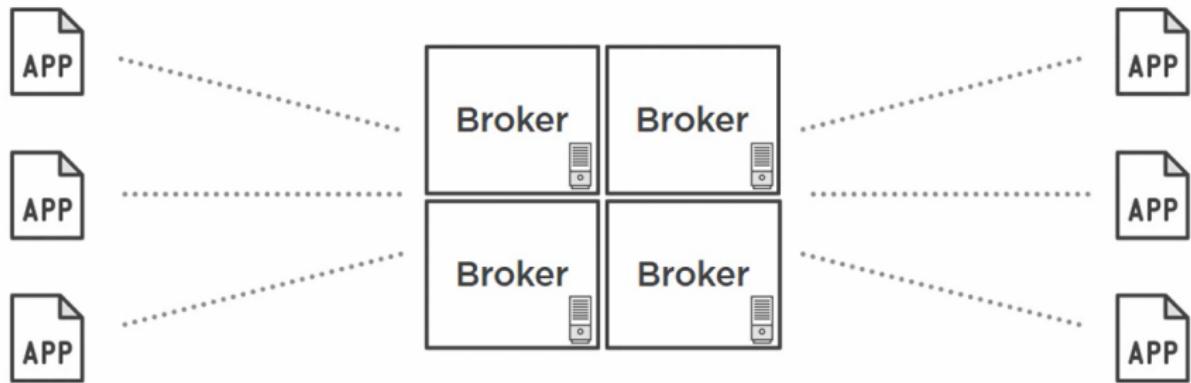


- Broker : gère un ensemble de topics
- C'est un processus (deamon) serveur

On peut créer un nombre quelconque de *brokers*

Producers

Consumers



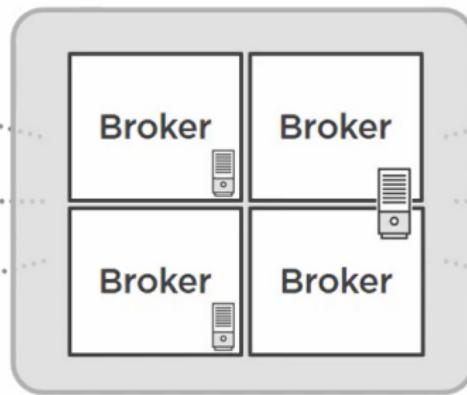
- C'est ce qui permet de gérer les gros débits de messages
- En 10.2019, LinkedIn a déclaré utiliser 4000 brokers avec 100 000 topics pour 7k milliards de messages par jour :
<https://engineering.linkedin.com/blog/2019/apache-kafka-trillion-messages>

Cluster

Producers



Cluster
Size: 4

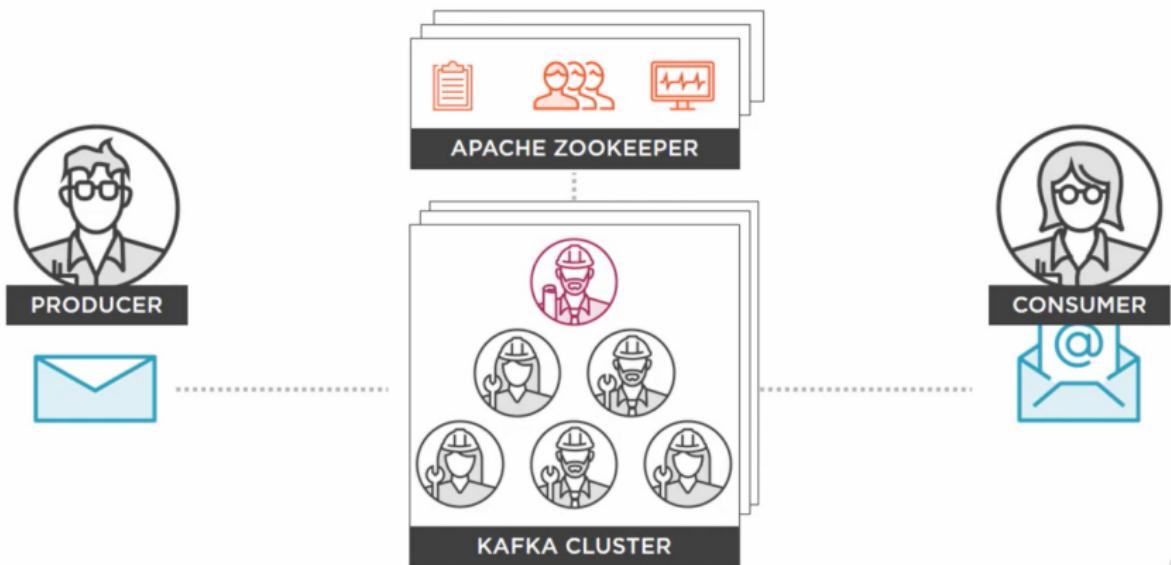


Consumers



- Cluster : ensemble de brokers

Apache Zookeeper



- Un système distribué (serveur) pour gérer les méta-données (état de santé, config, ...) sur les clusters
- C'est lui qui choisit quel broker est responsable de quel topic
- Utilisé dans d'autres projets aussi (Hadoop, Redis, Neo4j, ...)

Une intro différente à Kafka

- <https://www.youtube.com/watch?v=FKgi3n-FyNU>

Plan du cours

1. Introduction à Apache Kafka

2. Topics, Partitions et Brokers

3. Produire des messages

4. Consommer des messages

5. Conclusion

Installer Kafka à la main

- Ouvrir un terminal
- Installer Scala si vous ne l'avez pas (une partie de Kafka a été écrite avec Scala) : `sudo apt install scala` (sous Linux)
- Télécharger l'archive qui se trouve ici :

`wget`

`https://mirrors.ircam.fr/pub/apache/kafka/3.0.0/kafka_2.12-3.0.0.tgz`

Mettre à jour l'URL s'il y a une version plus récente

- Décompresser l'archive :

`tar xvzf kafka_2.12-2.6.0.tgz`

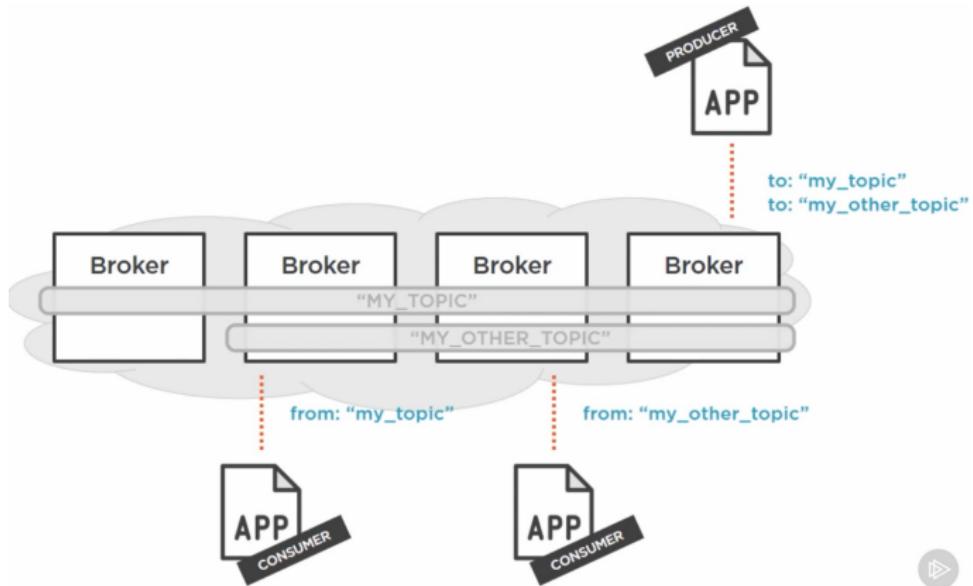
- Explorer le dossier, qui a une structure classique : `bin/` (scripts Shell + un dossier `windows/` avec les scripts Batch), `config/`, `libs/`, ...

Topics



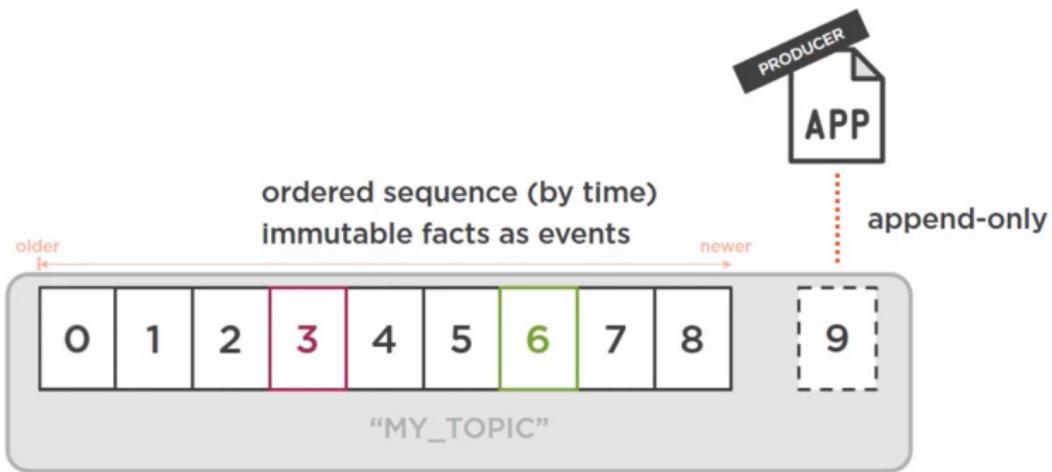
- Une abstraction (entité logique) au cœur du système
- Une catégorie ou un flux nommé de messages
- Les producteurs produisent des messages dans un topic
- Les consommateurs consomment des messages depuis un topic
- Physiquement représentés comme fichiers logs (Kafka = *distributed commit log*)

Topic : entité vue par les prod. & conso.



- Topics sur plusieurs brokers pour la fiabilité
- Mais les producteurs et consommateurs voient des topics (peu importe leur emplacement)

Topic : une séquence ordonnée

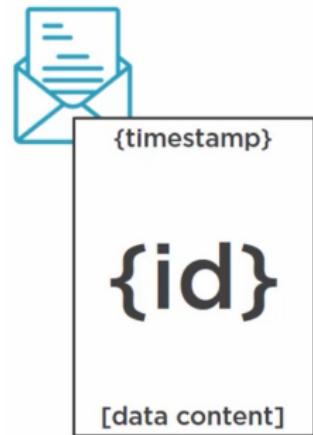


- Topic = séquence ordonnée par le temps
- Messages = événements (enregistrements ou *records*) immuables (système fonctionnant selon le style *Event Sourcing*)
- Messages insérés à la fin du topic et horodatés
- Au consommateur de repérer l'évolution d'un message (producteur incapable de changer un message existant)

Message

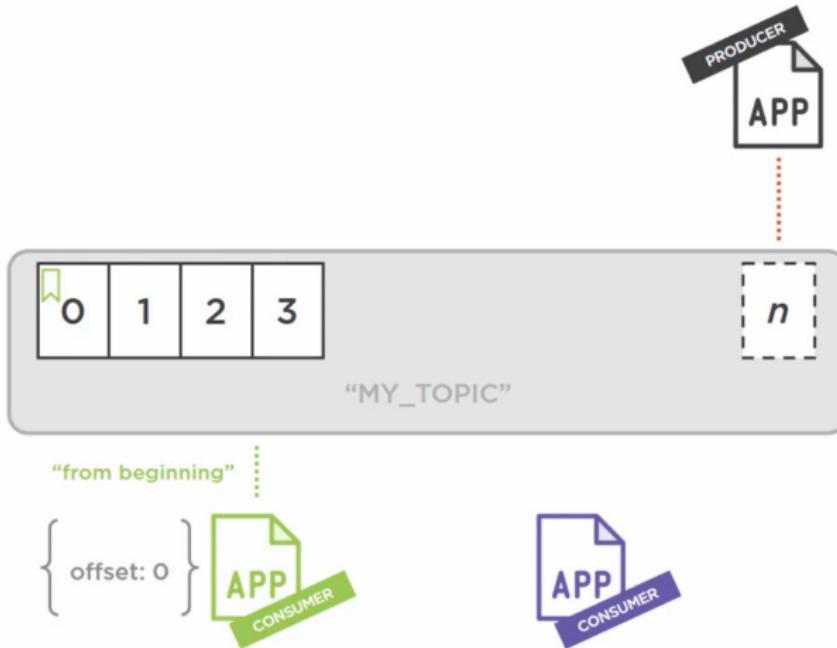
Chaque message :

1. est horodaté (a un *timestamp*)
2. a un identifiant unique utilisé par les producteurs et les consommateurs
La paire (*timestamp*,identifiant) est unique
3. a un contenu (*payload*) binaire



La défaillance d'un consommateur n'a aucun impact sur les messages, les autres consommateurs ou les producteurs

Offset



- Une sorte de marque-page qui correspond à la position du dernier message consommé
- Chaque consommateur peut évoluer indépendamment des autres dans la consommation des messages

Politique de rétention des messages

- Kafka garde tous les messages produits, qu'ils soient consommés ou pas
- La durée de rétention des messages est configurable en heures
- Par défaut, la durée de rétention est de 168 heures (7 jours)
- La période de rétention est définie par topic
- Le stockage physique peut contraindre la durée de rétention

Démo à faire, en ligne de commande

1. Démarrer un serveur Zookeeper :

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

* Un processus serveur est démarré sur le port 2181

2. Démarrer un broker :

```
bin/kafka-server-start.sh config/server.properties
```

* Un processus serveur est démarré sur le port 9092

3. Créer un topic :

```
bin/kafka-topics.sh --create --topic my_topic --zookeeper  
<votre-hostname>:2181 --replication-factor 1 --partitions 1
```

Pour obtenir le hostname, taper la commande hostname sur un terminal

4. Lister les topics :

```
bin/kafka-topics.sh --list --zookeeper <votre-hostname>:2181
```

Démo à faire, en ligne de commande -suite-

1. Produire un message :

```
bin/kafka-console-producer.sh --broker-list  
<votre-hostname>:9092 --topic my_topic
```

Ceci permet d'obtenir un Shell dans lequel on peut écrire plusieurs messages, séparés par des validations sur la touche "Entrée"

2. Consommer un message :

```
bin/kafka-console-consumer.sh --bootstrap-server  
<votre-hostname>:9092 --topic my_topic --from-beginning
```

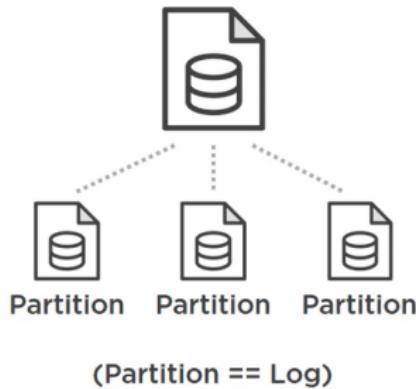
Les anciens messages s'affichent et chaque nouveau message produit par un producteur va apparaître

Pour supprimer un topic :

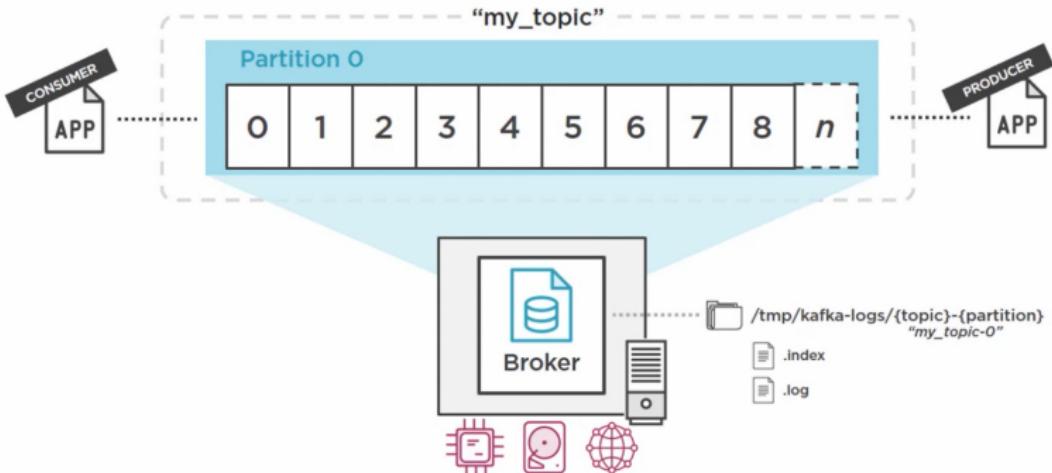
```
bin/kafka-topics.sh --delete --zookeeper <votre-hostname>:2181  
--topic my_topic
```

Partitions Kafka

- Une partition = un log physique (fichiers)
- Un topic a une ou +ieurs partitions
- Une partition est la base pour Kafka pour gérer la scalabilité, la tolérance aux fautes et la gestion d'un gros débit
- Chaque partition est maintenue sur au moins un broker
- Dans l'exemple précédent, nous avons créé un topic avec une seule partition (- -partitions 1)

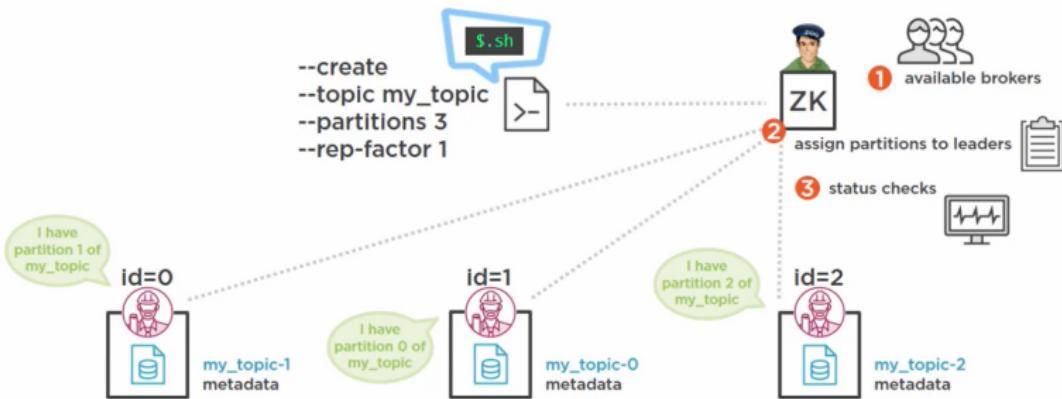


Une partition sur une machine



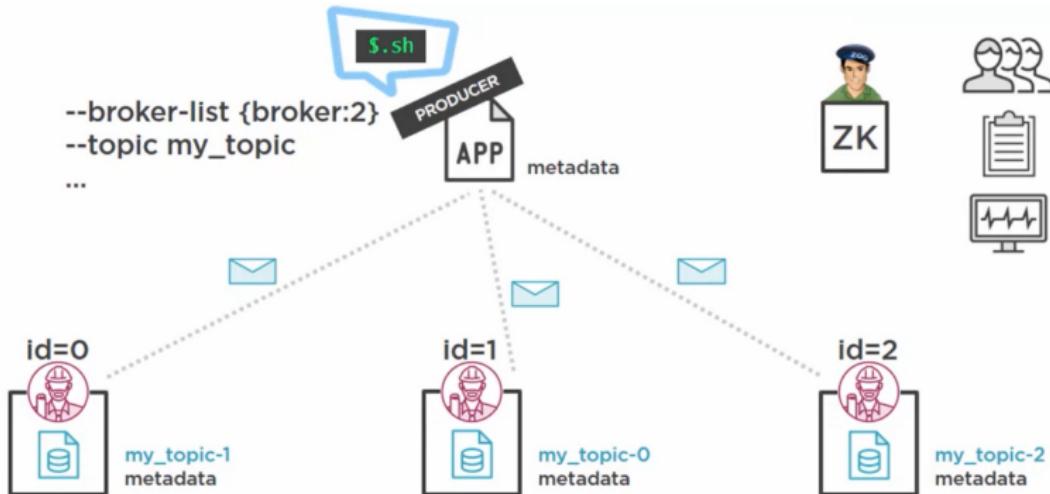
- Vérifier le contenu du dossier /tmp/kafka-logs
- Il y a un dossier pour votre topic, my_topic-0, avec des fichiers binaires .log (qui contient les messages), .index, ...
- Chaque partition doit tenir dans une machine

Plusieurs partitions pour la scalabilité



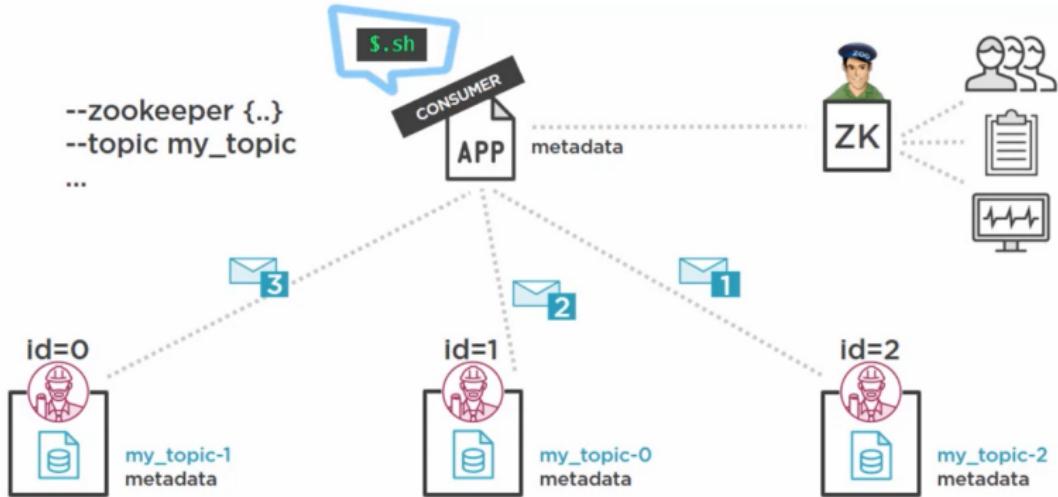
- Le serveur Zookeeper (ZK) gère l'affectation des partitions aux brokers
- Dans l'exemple, il va choisir 3 brokers, un pour chaque partition
- Les brokers remontent les informations sur leur état à ZK

Producteur de messages vers les brokers



- Un producteur de messages peut connaître 1 seul broker (broker 2 ci-dessus)
- Ce broker indique au producteur quels autres brokers détiennent les autres partitions (transparent pour le dev)
- La production du message envoie le message à tous les brokers

Consommateur de messages depuis les brokers

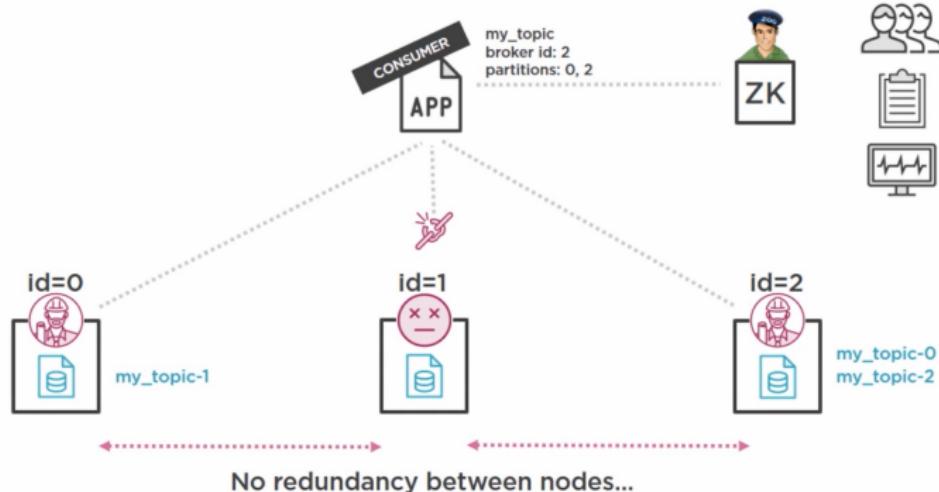


- Un consommateur collecte des méta-données depuis Zookeeper (liste et état des brokers)
- Il sollicite ensuite les différents brokers pour récupérer les messages
- Ces messages peuvent être ordonnés différemment sur chaque broker (le consommateur est responsable de gérer cela)

Compromis sur le nombre de partitions

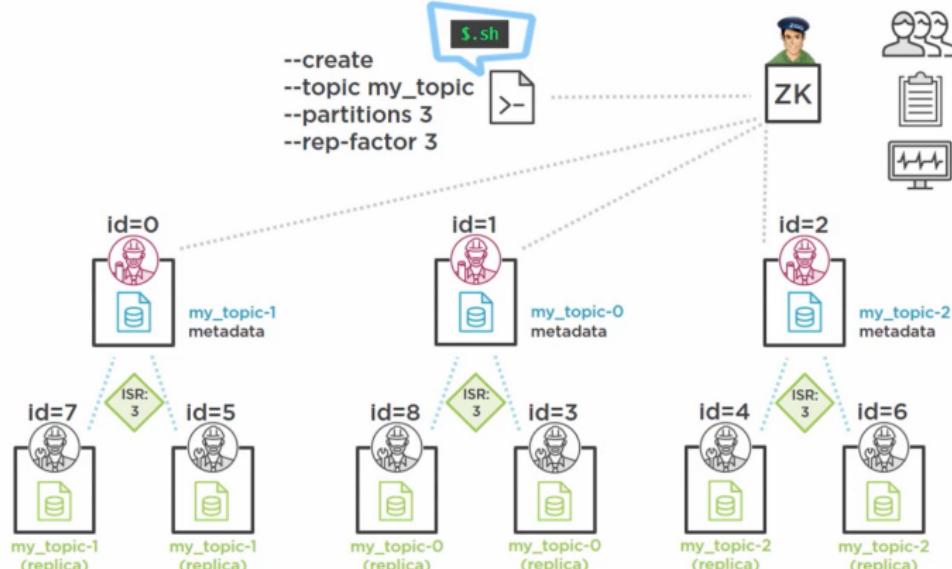
- Un nombre important de partitions a un impact négatif (surcharge) sur le Zookeeper (dans la coordination des brokers)
- Si l'on veut avoir un ordre total sur les messages, utiliser une seule partition par topic (sinon, ordres partiels / partition), ce qui peut être complexe à gérer dans certains cas

Tolérance aux fautes (ou aux pannes)



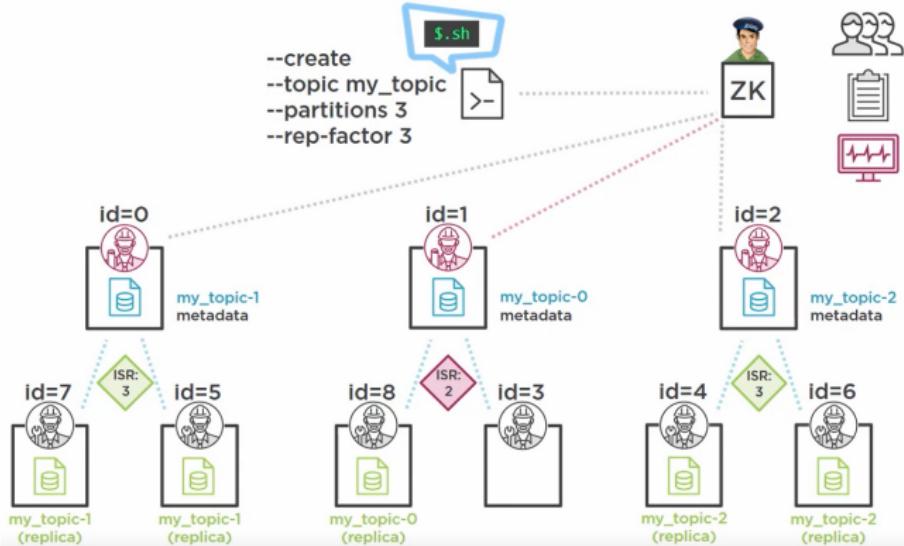
- Plusieurs fautes peuvent se produire (broker HS, coupure réseau, crash disque, ...)
- Le zookeeper gère l'inaccessibilité d'un broker pour informer les producteurs/consommateurs de messages (voir ci-haut)
- Mais les messages du broker en panne, qui n'ont pas été consommés, peuvent être perdus

RéPLICATION pour tolérer les fautes/pannes



- La tolérance aux fautes est atteinte par la réPLICATION
- Le paramètre replication-factor permet d'indiquer le nombre de répliqats **par topic**
- Chaque broker (nommé leader) négocie avec d'autres brokers la gestion de la réPLICATION

RéPLICATION pour tolérer les fautes/pannes -suite-



- *isr* veut dire *in-sync replicas*
- Tant que *isr* est égale à *replication-factor*, le cluster est considéré comme en bonne santé (*healthy cluster*)
- *isr < rep-factor* : Kafka continue de fonctionner mais ne fait pas de compensation

A vos claviers

Tester la création d'un cluster de 3 brokers :

- 3 exécutions de kafka-server-start avec 3 fichiers server.properties distincts (à copier, renommer et éditer)
server-0.properties, server-1.properties, ... avec :
 1. des id de brokers différents : 0, 1 et 2
 2. des dossiers différents pour les logs : /tmp/kafka-logs-0, /tmp/kafka-logs-1 ...
 3. et des ports différents : <votre-hostname>:9092, <votre-hostname>:9093, ...
- Créer un topic avec un facteur de réPLICATION = 3 :
`bin/kafka-topics.sh --create --topic replicated_topic
--zookeeper <votre-hostname>:2181 --replication-factor 3
--partitions 1`

A vos claviers -suite-

- Afficher les méta-données : (option --describe)

```
bin/kafka-topics.sh --describe --zookeeper  
<votre-hostname>:2181 --topic replicated_topic
```

- Ceci vous permet d'avoir :

```
Topic: replicated_topic  
PartitionCount: 1 ReplicationFactor: 3  
Configs:  
    Topic: replicated_topic Partition: 0 Leader: 1  
        Replicas: 1,0,2 Isr: 1,0,2
```

- Le leader de la réPLICATION est le broker dont l'id est 1 (brokers 0 et 2 participant)
- La ligne Topic aurait été affichée plusieurs fois si on avait plusieurs partitions

On peut altérer les méta-données d'un topic existant avec l'option --alter (pour ajouter par ex la réPLICATION à un cluster existant)

A vos claviers -suite-

- Créer un producteur de messages et produire quelques messages
- Créer un consommateur de messages
- Simuler une panne de l'un des brokers (le leader dont l'ID est 1), en tuant le processus de démarrage du broker (sur le terminal de démarrage de ce broker, faire un Ctrl-C ou un double Ctrl-C si vous êtes dans WSL sous Windows)
- Ré-exécuter la commande --descibe vue précédemment
- Que remarquez-vous dans le résultat affiché ?
- Le consommateur affiche une exception disant que le leader s'est arrêté, mais il continue de recevoir les messages
- Produire un message pour le vérifier
- Redémarrer le broker arrêté. Il va rejoindre le cluster. Pour le vérifier exécuter la commande --descibe

Plan du cours

1. Introduction à Apache Kafka
2. Topics, Partitions et Brokers
3. Produire des messages
4. Consommer des messages
5. Conclusion

Mettre en place l'environnement

- Un cluster avec au moins un broker doit être démarré
- Ajouter la dépendance suivante à votre projet :

```
implementation group: 'org.springframework.kafka', name: 'spring-kafka', version: '2.7.8'
```

- Il faudra définir le paramètre de config. suivant : (dans application.properties)

```
spring.kafka.bootstrap-servers=<votre-hostname>:9092
```

- Si vous avez des difficultés à accéder au cluster Kafka depuis votre app (dans la suite du cours), voir ici comment on peut démarrer un cluster (ZK et Kafka) avec docker-compose :
<http://wurstmeister.github.io/kafka-docker/>

Créer un topic Kafka

Écrire une classe de création de topics :

```
@Configuration
class KafkaTopicConfig {
    @Bean
    public NewTopic topic1() {
        return TopicBuilder.name("mytopic-1").build();
    }
    @Bean
    public NewTopic topic2() {
        return TopicBuilder.name("mytopic-2").build();
    }
    ...
}
```

On peut affiner la création de topics en indiquant, par ex, une période de rétention des messages (en millisecondes) :

```
<...>.config(TopicConfig.RETENTION_MS_CONFIG, "1680000").build();
```

Envoyer un message

```
@Component
class KafkaSenderExample {
    private KafkaTemplate<String , String > kafkaTemplate;
    @Autowired
    KafkaSenderExample(KafkaTemplate<String , String >
        kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }
    void sendMessage(String message , String topicName) {
        kafkaTemplate.send(topicName , message);
    }
}
```

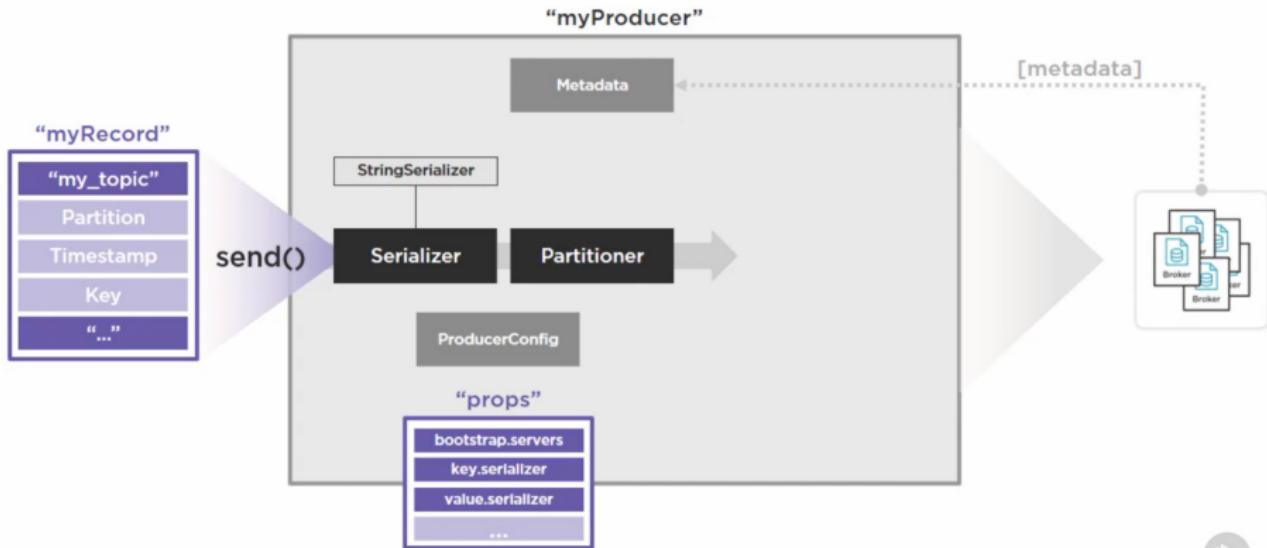
- Ici on utilise un String (De-)Serializer pour les clés/valeurs des messages (voir paramètres de KafkaTemplate)
- Ignorer le warning d'IntelliJ sur KafkaTemplate<String , String>
- Il est possible d'avoir un autre (de-)serializer pour JSON par ex :

<https://howtodoinjava.com/kafka/spring-boot-jsonserializer-example/>

Autres paramètres pour l'envoi de messages

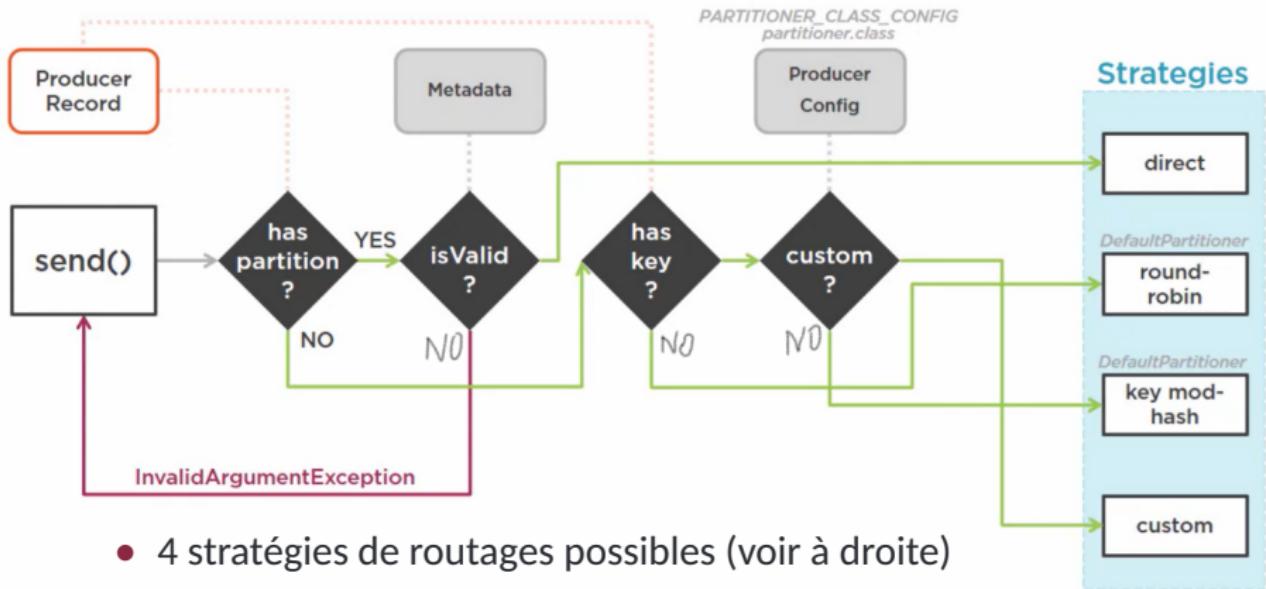
- L'objet kafkaTemplate a été utilisé dans cet exemple pour envoyer un message (une valeur, sans clé) à un topic
- Cet objet permet également d'envoyer des paires clés-valeurs plutôt que des valeurs seules (recommandé)
- Il permet aussi d'envoyer un message à une partition en particulier
- Plus généralement, on peut passer en paramètre un objet ProducerRecord (objet de base d'Apache Kafka, enveloppé par kafkaTemplate de Spring Kafka), qu'on peut paramétriser assez finement
- Plusieurs méthodes send(. . .) offertes par cet objet

Processus d'envoi de message – Partie 1



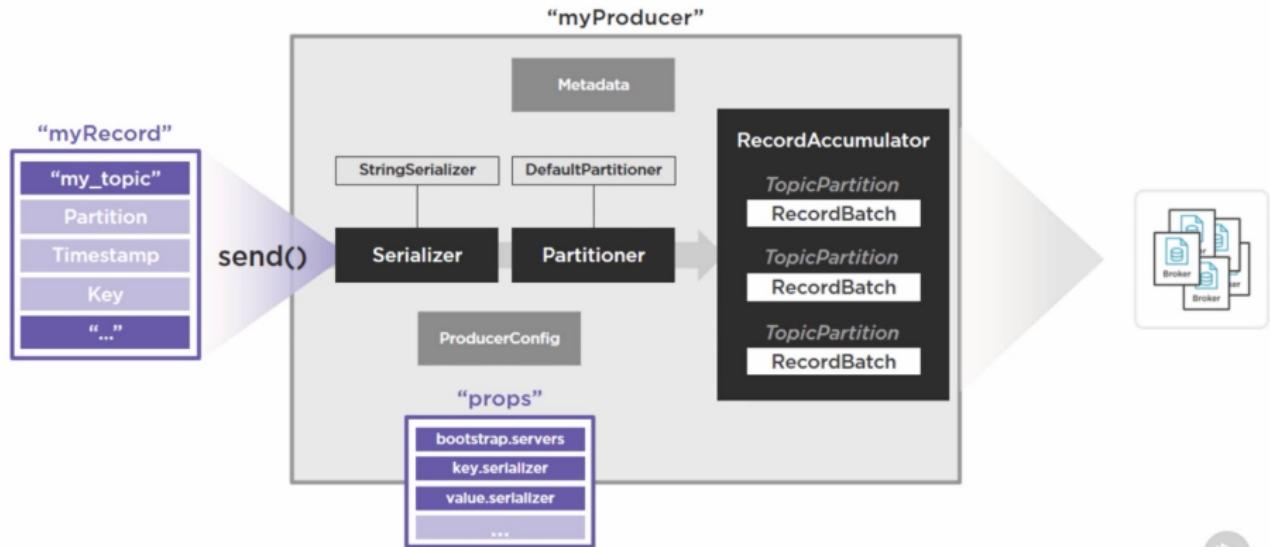
- L'envoi de messages est gérée par un objet KafkaProducer
- Cet objet va gérer la serialisation (vers String dans l'exemple) puis le routage vers les partitions

Routage vers les partitions



- 4 stratégies de routages possibles (voir à droite)
- De la plus simple (sans routage) au routage personnalisé défini dans une classe utilisateur

Processus d'envoi de message – Partie 2



- Dernière phase dans l'envoi de message est la bufferisation
- Plusieurs stratégies possibles (paramétrables) pour créer des batchs (cumuler plusieurs messages et faire un seul envoi, selon des contraintes de taille, de temps, ...)

Garanties d'envoi

- Il est possible d'indiquer des niveaux d'accusé de réception (acks) :
 - 0 : envoyer et oublier le message
 - 1 : accusé de réception du leader seulement
 - 2 : tous les ISR doivent envoyer un accusé de réception
- Il est possible également de faire des reprises d'envoi de messages (retry) et/ou forcer la reprise de l'envoi dans l'ordre (pas d'envoi d'un 2nd message tant que le 1er n'a pas réussi)
- Tout cela est paramétrable dans la configuration du *Producer*

A vos claviers

- Reprendre votre application Spring Boot et produire un message comportant le username et sa localisation, vers un topic dédié, non partitionné et répliqué avec un facteur de 3
- Utiliser un serializer String ("<username> :<localisation>") au début et ensuite un serializer JSON
- Vérifier la bonne réception des messages dans un consommateur lancé depuis le terminal, comme fait précédemment

Plan du cours

1. Introduction à Apache Kafka
2. Topics, Partitions et Brokers
3. Produire des messages
4. Consommer des messages
5. Conclusion

Même environnement que le producteur

- On utilise les mêmes paramètres que dans le producer (dans `application.properties`) : `bootstrap-servers`, ...
- La classe de base est `KafkaConsumer` qui permet à un objet de souscrire (*subscribe*) aux événements de réception de message
- Avec Spring MVC, c'est encore plus simple : une méthode ou une classe annotée `@KafkaListener`

Exemple de consommateur de messages

```
@Component
class KafkaListenerExample {
    Logger LOG = LoggerFactory.getLogger(KafkaListenerExample .
        class);
    @KafkaListener(topics = "my_topic")
    void listener(String data) {
        LOG.info(data);
    }
}
```

- Ici, on a une méthode qui est annotée par `@KafkaListener`
- Dans l'annotation, on ajoute le topic (ou les topics, séparés par virgules) au(x)quel(s) souscrit ce consommateur
- A la réception d'un message sur ce topic, la méthode est exécutée

Un consommateur de messages selon leur type

```
@Component
@Component(id = "class-level", topics = "my_topic")
class KafkaClassListener {
    @KafkaHandler
    void listen(String message) {
        LOG.info("KafkaHandler[ String ] {}", message);
    }
    @KafkaHandler(isDefault = true)
    void listenDefault(Object object) {
        LOG.info("KafkaHandler[ Default ] {}", object);
    }
}
```

- Ici, la méthode à exécuter est celle qui correspond au type spécifique du message reçu
- Si aucune méthode ne correspond au type de message reçu, celle marquée `isDefault` est exécutée

Un consommateur avec plus de paramètres

```
@KafkaListener(  
    groupId = "my_topic",  
    topicPartitions = @TopicPartition(  
        topic = "my_topic",  
        partitionOffsets = { @PartitionOffset(  
            partition = "0",  
            initialOffset = "0" ) } )  
void listenToPartitionWithOffset(  
    @Payload String message,  
    @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition,  
    @Header(KafkaHeaders.OFFSET) int offset) {  
    LOG.info("Received message [{}] from partition -{} with  
        offset -{}",  
    message,  
    partition,  
    offset);  
}
```

On verra la notion de groupe plus tard

Exemple précédent

- Dans l'exemple précédent, on a indiqué `initialOffset = "0"`
- Cela veut dire que le consommateur veut récupérer tous les messages depuis le début de la création de la partition (à chaque redémarrage de l'application)
- On a également utilisé l'annotation `@Header` qui a permis d'accéder à un certain nombre d'informations utiles : l'id de la partition et l'offset du message courant

Pour tester un consommateur avec une production "massive" de messages

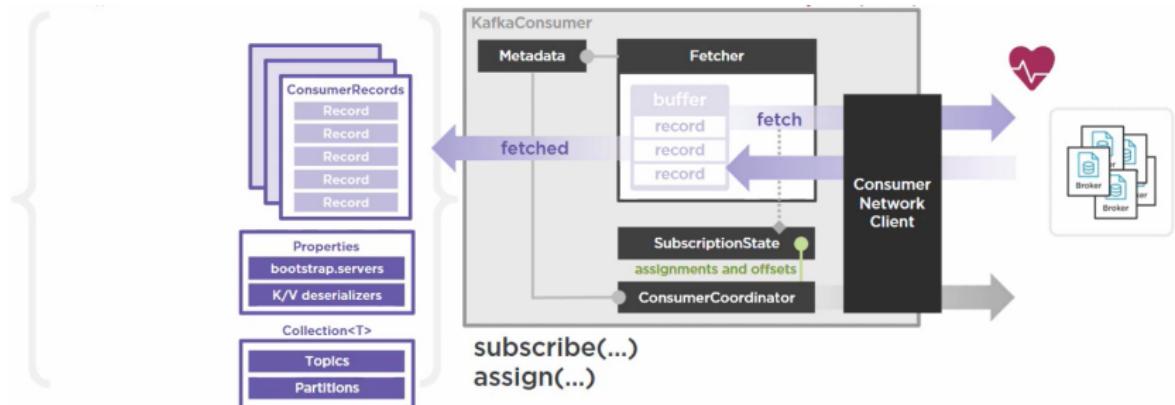
- Il existe un script qui vient avec Kafka et qui permet de produire une grande quantité de messages
- Celui-ci permet de tester les performances de la configuration
- Exécuter la commande :

```
bin/kafka-producer-perf-test.sh ---topic my_topic ---  
num-records 50 ---record-size 1 ---throughput 10  
---producer-props bootstrap.servers=<votre-hostname  
>:9092
```

Cette commande indique le topic cible, le nombre de messages (50), la taille des messages (1 octet), le débit (10 messages par seconde) et le broker

- A tester sur votre installation

Ce qui se cache derrière un consommateur



- L'objet au coeur des interactions avec le cluster est un objet de type KafkaConsumer (au centre de la figure)
- Celui-ci gère les méta-données, la souscription à des topics ou l'affectation (assign()) à des partitions de topics
- Il fait des polls périodiques (fréquence paramétrable) et bufferise les messages avant de les retourner au consommateur sous forme d'objets ConsumerRecord

Retour sur les offsets

- Un offset existe par partition et par consommateur
- Un message lu ne veut pas dire qu'il est validé (commit de son offset au broker). Il se peut que le consommateur tombe en panne pendant le traitement du message
- Par défaut, le consommateur fait un auto-commit (paramètre par défaut = true), lors de la lecture d'un message et après une durée de 5000 ms (paramètre par défaut), il envoie son dernier offset

Retour sur les offsets -suite-

- Il est possible de fixer la valeur de l'auto-commit à false. Dans application.properties :

```
enable.auto.commit=false  
spring.kafka.listener.ack-mode=manual
```

et ensuite faire des commits manuels pour éviter la perte de messages en cas de pannes :

- Ajouter un paramètre au listener :
Acknowledgment acknowledgment
 - Faire un commit : acknowledgment.acknowledge();
- Les offsets sont gérés par Kafka dans un topic à part nommé __consumer_offsets

Particularités des consommateurs

- Un consommateur donné effectue un poll dans un seul thread
- Si l'on fait un traitement lourd sur un message consommé, ceci n'a pas d'impact sur les autres composants du système (le cluster, les autres consommateurs ou les producteurs)
- Mais en pratique, il vaut mieux gérer les consommateurs par **groupe** pour se répartir la charge de consommation de messages
- C'est la solution de Kafka pour rendre les consommateurs scalables

Groupe de consommateurs

- Collection de consommateurs travaillant en équipe
- Pour qu'un consommateur puisse appartenir à un groupe, il faut définir le paramètre groupId
- Ils partagent la charge de consommation des messages
- **Une partition** est affectée à un consommateur d'un groupe
- S'il y a plus de consommateurs (d'un même groupe) que de partitions, les consommateurs en trop sont libres
- Dès qu'il y a une nouvelle partition qui se crée, elle est affectée au consommateur libre (*rebalancing*)
- Le *rebalancing* est fait aussi lorsque l'un des consommateur est arrêté pour attribuer sa/ses partition(s) aux autres consommateurs du groupe

Groupe de consommateurs -suite-

- Pour déclarer l'appartenance à un groupe, il suffit d'ajouter un *groupId* à un consommateur :

```
@KafkaListener(topics = "my_topic", groupId = "my-topic")
void listenToPartitionWithOffset2(
    ConsumerRecord<String, String> record,
    Acknowledgment acknowledgment) {
    LOG.info("Listener 2 - Received message [{}]
        from partition -{} with offset -{} recorded at: {}",
        record.value(),
        record.partition(),
        record.offset(),
        new Date(record.timestamp()));
    acknowledgment.acknowledge();
}
```

Noter la récupération du timestamp ci-dessus

Groupe de consommateurs -suite-

- Tester un groupe de consommateurs sur un topic à 3 partitions et observer quel consommateur/listener consomme les messages de quelle partition
 - Altérer le topic existant pour mettre en place de nouvelles partitions, si vous avez un topic à une partition :

```
bin/kafka-topics.sh --alter --zookeeper localhost:2181  
--topic my_topic --partitions 3
```
- Ajouter deux autres consommateurs et observez ce qui se passe
- Y a-t-il un consommateur (en trop) qui ne consomme rien ?
- Réduire le nombre de consommateurs à 2. A qui sont attribuées 2 partitions ?

Faire cela sans arrêter Zookeeper ou vos brokers. Exécuter les commandes dans un nouveau terminal, et modifier/ré-exécuter votre app Spring

Plan du cours

1. Introduction à Apache Kafka
2. Topics, Partitions et Brokers
3. Produire des messages
4. Consommer des messages
5. Conclusion

Wrap-up

- Kafka : un système distribué de gestion de messages à grand débit selon le modèle publish/subscribe
- On a vu comment mettre en place les trois composants de ce système : le cluster, les producteurs de messages et les consommateurs (avec Spring Kafka pour les deux derniers)
- De nombreux sujets n'ont pas été abordés ici : paramétrage fin du système (timeouts, délais, tailles des buffers, ...), contrôler les déplacements/flux de contrôle (`seek()`, `pause()`, `resume()`, ...), *Rebalance Listeners*, ..
- Plein de projets dans l'écosystème Kafka : *Schema Registry* (harmoniser les schémas de données et leur (dé-)serialisation), *Kafka Connect/Hub* (gérer l'hétérogénéité des sources/cibles de données), *Kafka Streams*, ... (chez confluent.io notamment)

Références bibliographiques

- Site web Apache Kafka :
<https://kafka.apache.org/>
- Site web spring.io :
<https://docs.spring.io/spring-kafka/docs/current/reference/html/>
- Tutos sur Pluralsight et Baeldung

