OXiane INSTITUT

make it **clever**

SOMMAIRE

SOMMAIRE

5. Schema registry

- Gestion des schéma (avro, json)
- API de manipulations

6. Streaming

- Introduction au concept de streaming et pipeline de données
- Comparatifs streams vs topics

7. KSQLDB

- Concepts et architecture de KSQLDB
- Requêtes KSQLDB et opérations en ligne de commande
- Traitement des données issues d'un stream
- Streams & Tables
- Jointures, agrégations et fenêtres de temps et de taille
- Développer une extension KSQLDB spécifique

Concepts fondamentaux

1. Producers sont des applications ou des services qui envoient des données à Kafka. Ils publient des **messages** dans des **topics**. Un topic est une catégorie ou un flux de données dans Kafka.

- Les producteurs envoient des messages dans des topics spécifiques.
- Ils peuvent partitionner les messages pour améliorer la distribution et la scalabilité.
- Les messages sont envoyés de manière synchrone ou asynchrone.

2. Consumers sont des applications ou des services qui lisent des données de Kafka. Ils s'abonnent à des topics et consomment les messages publiés par les producteurs.

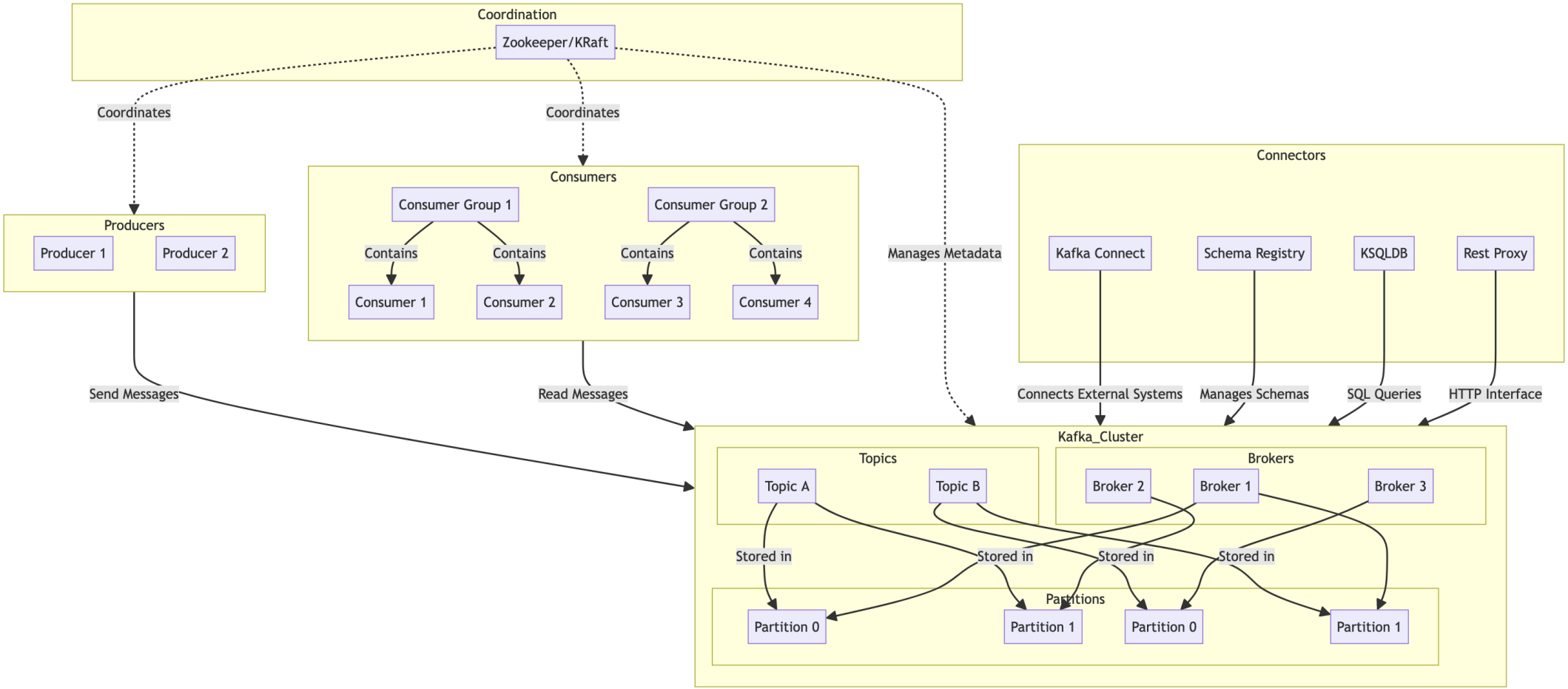
- Les consommateurs lisent les messages d'un topic.
- Chaque consommateur dans un **consumer group** lit des messages de partitions spécifiques, permettant une lecture parallèle.
- Les offsets des messages (positions de lecture) sont utilisés pour suivre les messages déjà consommés.

Concepts fondamentaux

3. Messages sont les unités de données envoyées par les producteurs à Kafka et lues par les consommateurs. Chaque message contient généralement une clé, une valeur et des métadonnées.

- Les messages sont publiés par les producteurs dans des topics.
- Les messages sont stockés de manière ordonnée dans des partitions.
- Chaque message dans une partition a un offset unique.

Architecture d'une plateforme Kafka



Architecture d'une plateforme Kafka

1. Kafka Cluster: Un cluster Kafka est constitué de plusieurs brokers (serveurs) qui stockent et gèrent les données.

- **Brokers** : Chaque broker est responsable de stocker les partitions de plusieurs topics. Les brokers communiquent entre eux pour la réplication des données et l'élection du leader.
- **Topics** : Les topics sont des flux de messages où les données sont publiées. Les topics sont divisés en partitions pour améliorer la scalabilité et la performance.
- **Partitions** : Les partitions sont des sous-ensembles d'un topic. Elles permettent de paralléliser la lecture et l'écriture des données pour améliorer la scalabilité.

2. Kafka Connect: Kafka Connect est un outil pour connecter Kafka avec des systèmes externes (bases de données, systèmes de fichiers, etc.).

- Kafka Connect utilise des connecteurs pour lire des données de systèmes externes et les publier dans des topics Kafka, ou pour lire des données de topics Kafka et les écrire dans des systèmes externes.

3. Schema Registry: Le Schema Registry est un service pour gérer les schémas de données (comme Avro, JSON Schema) utilisés dans Kafka.

- Le Schema Registry stocke et valide les schémas des messages. Les producteurs et consommateurs valident leurs messages contre ces schémas avant de les écrire ou de les lire de Kafka.

Architecture d'une plateforme Kafka

4. KSQLDB: KSQLDB est une base de données de streaming pour Kafka. Elle permet d'effectuer des requêtes SQL en temps réel sur les données en streaming.

- KSQLDB permet de créer des flux et des tables basées sur les données en temps réel dans Kafka, et de les interroger avec des requêtes SQL.

5. Rest Proxy: Le Rest Proxy fournit une interface HTTP pour interagir avec Kafka. Il permet de produire et consommer des messages via des requêtes REST.

- Les applications peuvent utiliser des requêtes HTTP pour envoyer et recevoir des messages de Kafka sans utiliser les bibliothèques clients Kafka natives.

6. Zookeeper/KRaft: Zookeeper était traditionnellement utilisé pour la gestion de la configuration et la coordination des brokers Kafka. KRaft est une alternative plus récente intégrée directement dans Kafka.

- Zookeeper gère la configuration du cluster, la coordination des brokers et des consommateurs, et le suivi des offsets. KRaft, introduit dans Kafka 2.8.0, élimine la dépendance à Zookeeper en intégrant ces fonctions directement dans Kafka.

Architecture d’une plateforme Kafka

Interactions entre les Composants

1. **Producers et Topics** : Les producteurs publient des messages dans des topics Kafka.
2. **Partitions et Brokers** : Les messages dans les topics sont stockés dans des partitions, et chaque partition est gérée par un broker spécifique.
3. **Consumers et Topics** : Les consommateurs s'abonnent aux topics et lisent les messages des partitions, en suivant les offsets pour savoir où ils en sont.
4. **Kafka Connect** : Connecte Kafka à des systèmes externes, permettant l'ingestion et l'exportation de données.
5. **Schema Registry** : Gère les schémas des messages pour assurer la compatibilité des données.
6. **KSQLDB** : Permet de traiter et d'interroger les données en temps réel avec SQL.
7. **Rest Proxy** : Offre une interface HTTP pour interagir avec Kafka.
8. **Zookeeper/KRaft** : Gère la configuration et la coordination des composants du cluster Kafka.

Développement pour Kafka

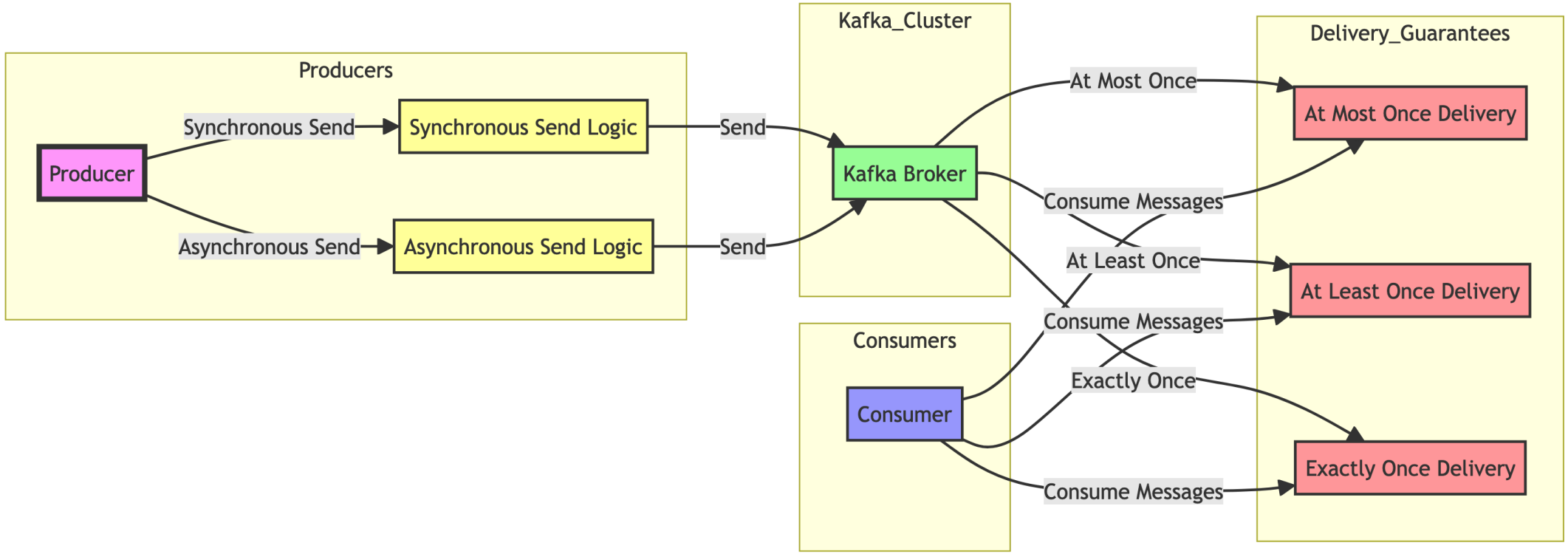
1. Java Development Kit (JDK) 8 ou supérieur
2. Apache Kafka
3. Maven ou Gradle
4. Spring Boot
5. Ajouter les dépendances Kafka :

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
    </dependency>
</dependencies>
```

6. Configurer Kafka dans Spring Boot :

```
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=my-group
spring.kafka.consumer.auto-offset-reset=earliest
```

Développement pour Kafka - Producer



Développement pour Kafka - Producer

1. Producers (Producteurs)

- **Synchronous Send** : Le producteur envoie un message et attend une confirmation avant de continuer. Cette méthode garantit que le message a été reçu par le broker.
- **Asynchronous Send** : Le producteur envoie un message sans attendre de confirmation immédiate. Il utilise des callbacks pour traiter les confirmations ou les erreurs de manière asynchrone.

2. Kafka Cluster

- Le broker Kafka reçoit les messages des producteurs. Il gère la persistance des messages dans les partitions des topics.

3. Delivery Guarantees (Garanties de Livraison)

- **At Most Once** : Les messages peuvent être perdus mais ne sont jamais doublés. Utilisé lorsque la perte de quelques messages est acceptable.
- **At Least Once** : Les messages sont toujours livrés, mais peuvent être doublés en cas de réessai. Utilisé lorsque la duplication est acceptable mais la perte de messages ne l'est pas.
- **Exactly Once** : Les messages sont livrés exactement une fois sans perte ni duplication. C'est la garantie la plus stricte, souvent utilisée dans les transactions critiques.

Développement pour Kafka - Producer

1. **KafkaTemplate**: `KafkaTemplate` est l'objet principal utilisé pour l'envoi de messages dans Kafka. Il est fourni par Spring Kafka et offre des méthodes pour envoyer des messages de manière synchrone et asynchrone.
2. **ProducerFactory**: `ProducerFactory` est utilisé pour créer des instances de `KafkaProducer`. Il configure les propriétés du producteur, telles que les serveurs bootstrap, les sérialiseurs, etc.
3. **ProducerRecord**: `ProducerRecord` est un objet représentant un message Kafka à envoyer. Il contient les informations du topic, de la clé (facultative), et de la valeur.
4. **SendResult**: `SendResult` est l'objet retourné après l'envoi d'un message, contenant des informations sur le message envoyé, telles que le topic, la partition, et l'offset.

```
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.acks=all
spring.kafka.producer.retries=10
spring.kafka.producer.batch-size=16384
spring.kafka.producer.linger-ms=1
spring.kafka.producer.buffer-memory=33554432
```

Développement pour Kafka - Producer

1. Production Synchrone

Dans la production synchrone, le producteur attend une confirmation de Kafka avant de continuer. Cela garantit que le message a été reçu par Kafka.

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.support.SendResult;
import org.springframework.stereotype.Service;
import org.springframework.util.concurrent.ListenableFuture;

import java.util.concurrent.ExecutionException;

@Service
public class SynchronousProducer {

    private final KafkaTemplate<String, String> kafkaTemplate;

    public SynchronousProducer(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void sendMessage(String topic, String message) {
        ListenableFuture<SendResult<String, String>> future = kafkaTemplate.send(topic, message);
        try {
            SendResult<String, String> result = future.get();
            System.out.printf("Sent message to topic %s partition %d offset %d%n",
                    result.getRecordMetadata().topic(),
                    result.getRecordMetadata().partition(),
                    result.getRecordMetadata().offset());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

Développement pour Kafka - Producer

2. Production Asynchrone

Dans la production asynchrone, le producteur envoie un message sans attendre de confirmation immédiate. Un callback est utilisé pour traiter la confirmation ou les erreurs de manière asynchrone.

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.support.SendResult;
import org.springframework.stereotype.Service;
import org.springframework.util.concurrent.ListenableFuture;
import org.springframework.util.concurrent.ListenableFutureCallback;

@Service
public class AsynchronousProducer {

    private final KafkaTemplate<String, String> kafkaTemplate;

    public AsynchronousProducer(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void sendMessage(String topic, String message) {
        ListenableFuture<SendResult<String, String>> future = kafkaTemplate.send(topic, message);

        future.addCallback(new ListenableFutureCallback<SendResult<String, String>>() {
            @Override
            public void onSuccess(SendResult<String, String> result) {
                System.out.printf("Sent message to topic %s partition %d offset %d%n",
                        result.getRecordMetadata().topic(),
                        result.getRecordMetadata().partition(),
                        result.getRecordMetadata().offset());
            }
            @Override
            public void onFailure(Throwable ex) {
                System.err.println("Failed to send message: " + ex.getMessage());
            }
        });
    }
}
```

Développement pour Kafka - Producer

3. Production avec Accusés de Réception

Pour garantir que les messages sont reçus par tous les réplicas des partitions, vous pouvez configurer `acks=all`.

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.support.SendResult;
import org.springframework.stereotype.Service;
import org.springframework.util.concurrent.ListenableFuture;

import java.util.concurrent.ExecutionException;

@Service
public class AckProducer {

    private final KafkaTemplate<String, String> kafkaTemplate;

    public AckProducer(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void sendMessage(String topic, String message) {
        ListenableFuture<SendResult<String, String>> future = kafkaTemplate.send(topic, message);
        try {
            SendResult<String, String> result = future.get();
            System.out.printf("Sent message to topic %s partition %d offset %d%n",
                    result.getRecordMetadata().topic(),
                    result.getRecordMetadata().partition(),
                    result.getRecordMetadata().offset());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

Développement pour Kafka - Producer

4. Production avec Transactions

Pour garantir l'atomicité des messages envoyés, vous pouvez utiliser les transactions Kafka.

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class TransactionalProducer {

    private final KafkaTemplate<String, String> kafkaTemplate;

    public TransactionalProducer(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
        this.kafkaTemplate.setTransactionIdPrefix("txn-");
    }

    @Transactional
    public void sendMessage(String topic, String message) {
        kafkaTemplate.executeInTransaction(operations -> {
            operations.send(topic, message);
            return true;
        });
    }
}
```

Développement pour Kafka - Producer

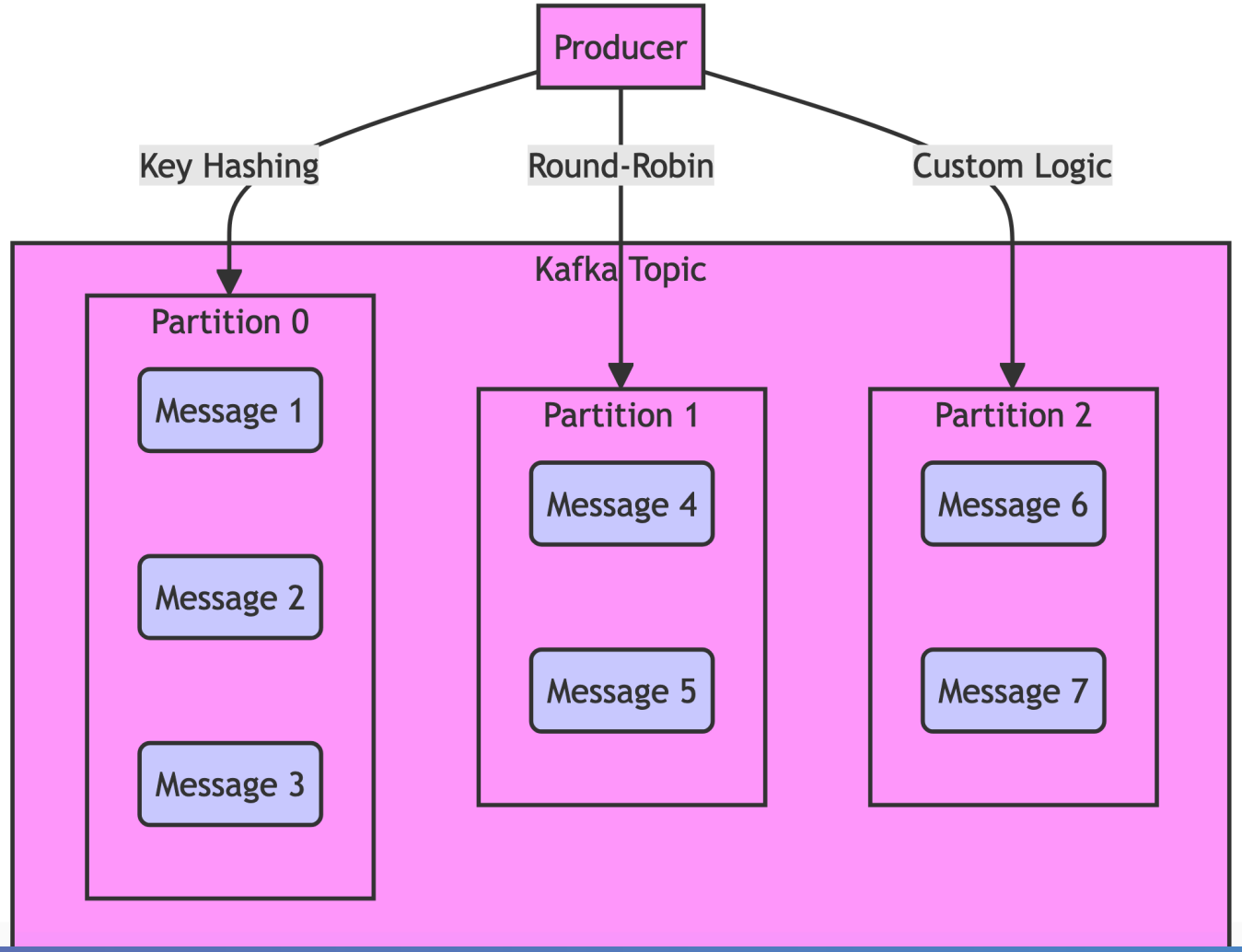
- **spring.kafka.bootstrap-servers** : Spécifie les adresses des serveurs Kafka à utiliser.
- **spring.kafka.producer.key-serializer** : Sérialiseur de clé pour le producteur.
- **spring.kafka.producer.value-serializer** : Sérialiseur de valeur pour le producteur.
- **spring.kafka.producer.acks** : Détermine le niveau d'accusés de réception souhaité. `all` garantit que tous les réplicas reconnaissent le message.
- **spring.kafka.producer.retries** : Nombre de tentatives de réessai en cas d'échec de l'envoi.
- **spring.kafka.producer.batch-size** : Taille des lots de messages à envoyer.
- **spring.kafka.producer.linger-ms** : Temps d'attente avant d'envoyer un lot de messages.
- **spring.kafka.producer.buffer-memory** : Taille de la mémoire tampon pour le producteur.

Exercice 1

Vous travaillez pour une entreprise qui développe un système de notifications pour un réseau social. Les notifications peuvent être envoyées pour divers événements tels que des likes, des commentaires, ou des nouveaux messages. Pour garantir la réactivité et la scalabilité du système, vous allez utiliser Apache Kafka pour gérer ces notifications. Vous devez implémenter deux types de producteurs Kafka : un producteur synchrone pour les notifications critiques (comme les nouveaux messages) et un producteur asynchrone pour les notifications moins critiques (comme les likes).

1. **Implémentation du Producteur Synchrone**
   - Implémentez un producteur Kafka synchrone pour envoyer des notifications critiques.
   - Assurez-vous que le producteur attend la confirmation de Kafka avant de continuer.
2. **Implémentation du Producteur Asynchrone**
   - Implémentez un producteur Kafka asynchrone pour envoyer des notifications générales.
   - Utilisez des callbacks pour gérer les confirmations ou les échecs de manière asynchrone.
3. **Test des Producteurs**
   - Créez des endpoints REST pour tester l'envoi de messages via les producteurs synchrones et asynchrones.
   - Envoyez des notifications critiques et générales et observez les comportements.

Partitionnement



Partitionnement

Dans Kafka, un "topic" est une catégorie ou un flux de messages. Pour permettre la scalabilité horizontale et une meilleure performance de lecture/écriture, chaque topic peut être divisé en plusieurs "partitions". Chaque partition est, en essence, un fichier journal immuable où les messages sont écrits de manière séquentielle.

Avantages du partitionnement :

- **Parallélisme** : Les partitions permettent à plusieurs consommateurs de lire les messages en parallèle, augmentant ainsi la capacité de traitement du système.
- **Réplication** : Chaque partition peut être répliquée sur plusieurs nœuds pour assurer la haute disponibilité et la durabilité des données.
- **Équilibrage de charge** : Distribuer les messages sur plusieurs partitions peut aider à équilibrer la charge entre différents serveurs.

Comment fonctionne le partitionnement ?

- Lorsque vous produisez un message, vous pouvez spécifier une clé. Kafka utilise cette clé pour attribuer le message à une partition spécifique en utilisant une fonction de hachage, ce qui garantit que tous les messages avec la même clé vont à la même partition.
- Si aucune clé n'est spécifiée, Kafka distribue les messages de manière round-robin entre toutes les partitions disponibles du topic, assurant ainsi une distribution équilibrée des messages.

Partitionnement personnalisé dans Apache Kafka

Le partitionnement personnalisé vous permet de contrôler la logique de distribution des messages à des partitions spécifiques, basée sur des critères autres que le simple hachage de clés ou le round-robin. Cela peut être utile pour des raisons de performances ou pour des exigences métier spécifiques.

Implémentation avec Kafka :

1. **Créer une classe de partitionneur personnalisé** : Implémentez l'interface `Partitioner` dans votre code. Cette interface requiert la définition de la méthode `partition()`, où vous spécifiez comment les messages sont attribués aux partitions.
2. **Configurer le producteur** : Lorsque vous configurez votre producteur Kafka, spécifiez votre classe de partitionneur personnalisée en utilisant la propriété `partitioner.class`.

Exercice 2: Gestion des Partitions

- Intégrer une gestion personnalisée des partitions pour diriger les notifications vers des partitions spécifiques en fonction de leur type et de leur importance.

1. **Définition des Règles de Partitionnement**
   - Définissez des règles pour partitionner les messages envoyés par les producteurs. Par exemple, toutes les notifications de "nouveaux messages" peuvent aller à une partition spécifique, tandis que les "likes" peuvent être répartis de manière équilibrée entre les autres partitions.
   - Implémentez une classe de partitionnement personnalisée dans Kafka pour appliquer ces règles.
2. **Modification des Producteurs pour Utiliser le Partitionnement Personnalisé**
   - Modifiez le producteur synchrone et asynchrone pour utiliser votre partitionneur personnalisé. Assurez-vous que le type de notification influence la partition à laquelle le message est envoyé.
   - Pour le producteur synchrone, garantissez que les notifications critiques sont envoyées à des partitions qui sont moins susceptibles de surcharger, pour assurer une livraison rapide.
   - Pour le producteur asynchrone, utilisez un partitionnement qui optimise l'utilisation des ressources et équilibre la charge entre les partitions.

Développer un consumer pour s’abonner à un topic kafka

Dans Apache Kafka, les consommateurs jouent un rôle crucial dans le traitement et la consommation des messages stockés dans les topics. Kafka offre des mécanismes flexibles pour que les consommateurs récupèrent les messages, principalement à travers deux modes de consommation : **subscribe** (s'abonner) et **assign** (attribuer).

1. Subscribe (S'abonner)

Quand des consommateurs utilisent `subscribe`, ils s'abonnent à un ou plusieurs topics. Cela signifie qu'ils indiquent à Kafka leur intention de recevoir des messages de ces topics spécifiques. Les caractéristiques de ce mode incluent :

- **Groupe de Consommateurs** : Les consommateurs qui s'abonnent à un topic sont généralement organisés en groupes de consommateurs. Kafka garantit que chaque partition d'un topic est consommée par un seul membre du groupe à la fois, ce qui permet une consommation équilibrée et une haute disponibilité.
- **Rééquilibrage Automatique** : Si un nouveau consommateur rejoint le groupe ou si un consommateur existant quitte le groupe ou échoue, Kafka réorganise automatiquement les partitions parmi les consommateurs actifs du groupe. Cela s'assure que toutes les partitions restent consommées et aide à éviter des points de défaillance.

Développer un consumer pour s’abonner à un topic kafka

2. Assign (Attribuer)

Dans le mode `assign`, les consommateurs spécifient explicitement les partitions des topics qu'ils souhaitent consommer, sans utiliser le concept de groupe de consommateurs. Les points clés ici sont :

- **Contrôle Granulaire** : Cette méthode donne aux consommateurs un contrôle précis sur les partitions à consommer, ce qui peut être utile pour des cas d'utilisation avancés où un rééquilibrage automatique n'est pas souhaité ou nécessaire.
- **Pas de Rééquilibrage** : Il n'y a pas de rééquilibrage automatique des partitions comme avec `subscribe`. Le consommateur est entièrement responsable de la gestion des partitions qui lui sont attribuées.

Développer un consumer pour s’abonner à un topic kafka

- **Partitionnement Efficace** : Avoir plus de partitions permet une parallélisation accrue, mais aussi peut augmenter le temps de latence et les coûts de gestion. Le choix du nombre de partitions peut affecter significativement les performances et la scalabilité.
- **Facteur de Réplication** : Chaque partition peut être répliquée sur plusieurs serveurs pour garantir la durabilité et la haute disponibilité des données.

3. Mécanisme de Commit dans Kafka

Kafka maintient ce qu'on appelle un "offset" pour chaque consommateur dans chaque partition. L'offset est un pointeur qui indique la position du dernier message consommé dans la partition. Quand un consommateur lit des messages d'une partition, il avance cet offset message par message.

Le commit de l'offset permet de:

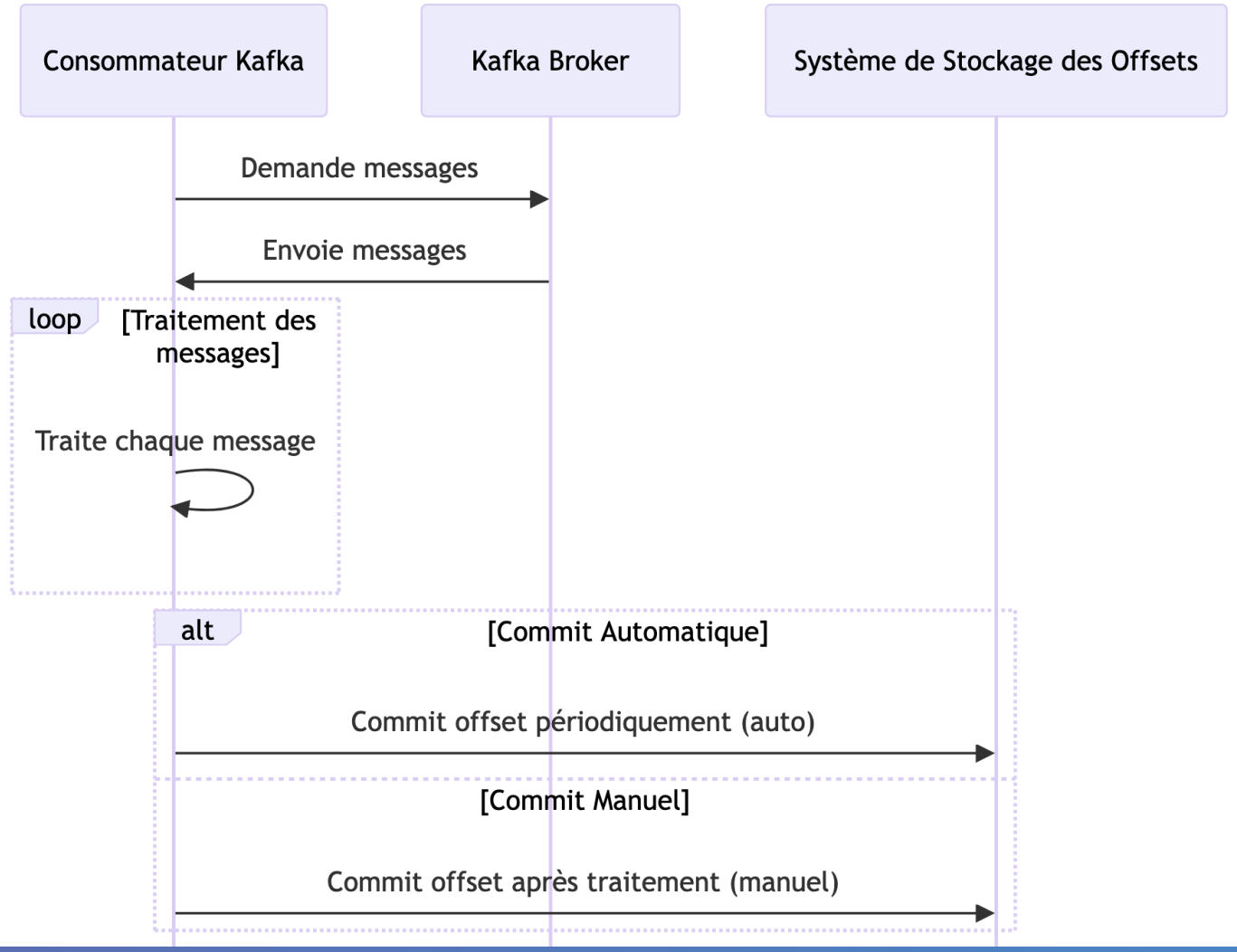
1. **Assurer la fiabilité** : En cas de panne du consommateur ou de redémarrage du système, Kafka sait à partir de quel message recommencer la lecture, grâce aux offsets committés. Cela garantit que les messages ne sont pas perdus ou consommés en double.
2. **Contrôle de la consommation** : Les consommateurs peuvent gérer la vitesse à laquelle ils consomment les messages et s'assurer qu'ils ne procèdent au commit que lorsque le traitement des messages est terminé.

Développer un consumer pour s’abonner à un topic kafka

Kafka offre deux modes principaux de commit d'offsets :

1. **Commit Automatique** :
   - **Configuration** : Cela est contrôlé par la propriété de configuration `enable.auto.commit`. Si elle est définie sur `true`, Kafka committera automatiquement les offsets à intervalles réguliers spécifiés par la propriété `auto.commit.interval.ms`.
   - **Utilisation** : Ce mode est plus simple à utiliser mais peut mener à des situations où des messages sont considérés comme consommés alors qu'ils ne l'ont pas été correctement traités en cas de panne juste avant un traitement mais après un commit.
2. **Commit Manuel** :
   - **Contrôle** : Le consommateur appelle explicitement une méthode pour committer l'offset après avoir traité les messages. Cela peut être fait de manière synchrone (attendre la réponse de Kafka confirmant le commit) ou asynchrone (ne pas attendre la réponse).
   - **Utilisation** : Ce mode offre un meilleur contrôle et est généralement utilisé dans les applications où la précision de la consommation et la garantie du traitement des messages sont critiques.

Développer un consumer pour s'abonner à un topic kafka



Exercice 3

Configurer des consommateurs Kafka pour traiter les notifications de manière efficace en fonction du type de notification et de la partition à partir de laquelle ils consomment.

1. Groupes de Consommateurs Spécialisés

- Configurez différents groupes de consommateurs pour différents types de notifications. Par exemple, un groupe peut être dédié aux notifications critiques (“nouveaux messages”) et un autre aux notifications moins critiques (“likes”).
- Assurez-vous que chaque groupe est optimisé pour traiter le type de notifications qui lui est attribué.

2. Configuration des Consommateurs pour Gérer Différentes Partitions

- Chaque groupe de consommateurs doit être configuré pour consommer des partitions spécifiques, conformément à la stratégie de partitionnement définie précédemment.
- Utilisez la méthode assign pour un contrôle précis des partitions ou subscribe pour une gestion plus dynamique et automatisée avec des rééquilibrages.

3. Traitement Basé sur le Type de Notification

- Implémentez une logique dans les consommateurs pour traiter différemment les messages en fonction de leur type. Les notifications critiques peuvent nécessiter des actions immédiates, tandis que les autres peuvent être traitées de manière plus asynchrone.

Kafka Connect

Kafka Connect est un projet open-source qui fournit un cadre scalable et fiable pour intégrer Kafka avec des systèmes externes. Il simplifie le processus de création et de gestion des pipelines de données entre Kafka et diverses sources ou destinations de données, permettant une intégration de données et un traitement de flux transparents. Kafka Connect sert d'outil puissant pour les développeurs, les ingénieurs de données et les architectes qui doivent gérer efficacement les tâches d'ingestion et de réplication de données.

- **Intégration unifiée des systèmes de données en streaming et par lots** : Kafka Connect est une solution idéale pour faire le pont entre les systèmes de données en streaming et par lots. Il permet aux développeurs de construire des pipelines de données intégrant Kafka avec d'autres systèmes de données, tels que les bases de données relationnelles, les bases de données NoSQL, les magasins d'objets et les systèmes de fichiers.
- **Un cadre commun pour les connecteurs Kafka** : Kafka Connect standardise l'intégration d'autres systèmes de données avec Kafka, simplifiant le développement, le déploiement et la gestion des connecteurs.
- **Extensibilité** : Kafka Connect offre une architecture de plugin qui permet aux développeurs d'étendre ses fonctionnalités en construisant des connecteurs personnalisés. Cette extensibilité permet une intégration transparente avec de nouvelles sources ou destinations de données, permettant aux organisations de s'adapter et d'évoluer dans leurs pipelines de données à mesure que leurs besoins changent.
- **Facilité de gestion** : Nous pouvons soumettre et gérer les connecteurs pour le cluster Kafka Connect via une API REST facile à utiliser.
- **Modes distribué et autonome** : Nous pouvons mettre à l'échelle jusqu'à de grands clusters (avec le mode distribué) ou réduire à des déploiements de développement, de test ou de petite production (avec le mode autonome).

Kafka Connect

Kafka Connect prend en charge deux modes d'exécution :

- **Mode autonome (processus unique)**
- **Mode distribué**

Dans le mode autonome, tout le travail est effectué dans un seul processus. Cette configuration est plus simple à configurer et à démarrer et peut être utile dans des situations où un seul travailleur est applicable (par exemple, la collecte de fichiers journaux), mais elle ne bénéficie pas de certaines fonctionnalités de Kafka Connect, telles que la tolérance aux pannes.
Dans le mode distribué, Kafka Connect fonctionne comme un système distribué avec plusieurs instances de travailleurs exécutant sur différents nœuds. Il gère l'équilibrage automatique du travail, nous permet de mettre à l'échelle dynamiquement et offre une tolérance aux pannes à la fois dans les tâches actives et pour la configuration des données de compensation engagées. En mode distribué, Kafka Connect stocke les décalages, les configurations et les statuts des tâches dans des topics Kafka.

Kafka Connect

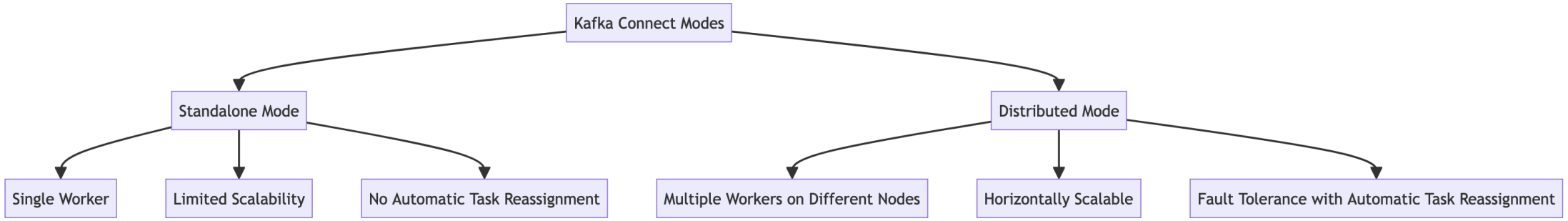
Caractéristique	Mode Distribué	Mode Autonome
Déploiement	Plusieurs travailleurs sur différents nœuds	Un seul travailleur sur une seule machine
Scalabilité	Scalable horizontalement	Scalabilité limitée
Tolérance aux pannes	Tolérant aux pannes avec réaffectation automatique des tâches	Pas de réaffectation automatique des tâches
Équilibrage de charge	Charge de travail automatiquement équilibrée parmi les travailleurs	Pas d'équilibrage de charge
Cas d'utilisation	Déploiements à grande échelle, volumes de données élevés	Développement, tests, déploiements à petite échelle

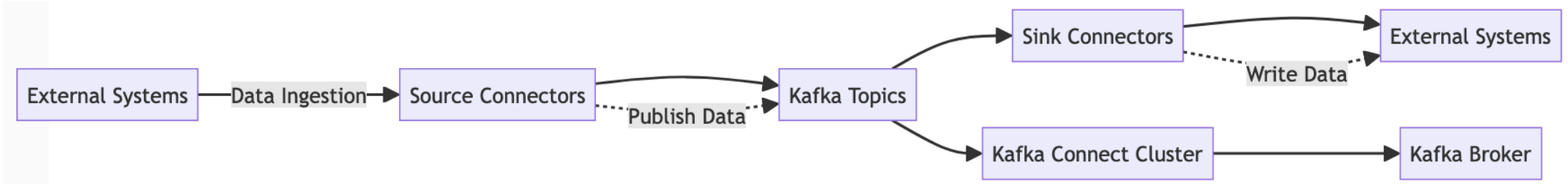
Kafka Connect

Les connecteurs Kafka Connect permettent l'intégration entre Apache Kafka et des systèmes externes pour l'ingestion et la réplication de données. Ce sont des modules enfichables qui définissent la logique et la configuration nécessaires pour lire les données de ou les écrire vers des sources ou des puits de données spécifiques. En utilisant des connecteurs, les développeurs, les ingénieurs de données et les architectes peuvent construire et gérer efficacement des pipelines de données qui interagissent avec diverses sources telles que des bases de données, des files d'attente de messages, des fichiers journaux et des plateformes de médias sociaux.

Les connecteurs gèrent les subtilités de l'extraction des données de ces sources ou de leur livraison, en abstrayant les complexités de l'intégration avec différents systèmes. Cette abstraction permet aux utilisateurs de se concentrer sur la configuration du pipeline et d'exploiter la puissance de la plateforme de messagerie distribuée de Kafka, plutôt que de traiter les spécificités de chaque système externe.

Kafka Connect





Exercice 4: Intégration et Gestion des Connecteurs Kafka dans Spring Boot

Intégrer et gérer les connecteurs Kafka Connect pour synchroniser les données entre les systèmes de notifications et les plateformes d'analyse ou de stockage externe, en utilisant Spring Boot pour le contrôle et l'orchestration.

1. Configuration des Connecteurs Kafka dans Spring Boot

- Configurez des connecteurs source pour extraire les notifications à partir de systèmes de bases de données.
- Configurez des connecteurs sink pour envoyer des notifications traitées vers des systèmes comme Elasticsearch pour l'analyse, ou vers des systèmes de stockage comme Amazon S3.

2. Développement de l'API pour Gérer les Connecteurs

- Créez une API RESTful dans Spring Boot pour ajouter, modifier, et supprimer des configurations de connecteurs en direct.