Ateliers
« Spring / Kafka »

Pré-requis :

- Bonne connexion Internet
- Système d'exploitation : Linux / Windows / MacOS
- JDK21+, Maven
- IDE Recommandés : STS 4, IntelliJIDEA, VSCode avec support pour Lombok
- Docker, Git
- Apache JMeter

Dépôt des supports :
https://github.com/dthibau/springkafka

Dans le répertoire ***TPS***, des sous-répertoires numérotés fournissent les fichiers sources nécessaires pour les ateliers.

Dépôt des solutions :
https://github.com/dthibau/springkafka-solutions

Le dépôt contient des tags correspondants aux ateliers.

Images docker utilisées :

- *docker.io/bitnami/kafka:3.7*
- *docker.redpanda.com/redpandadata/console:latest*
- *confluentinc/cp-schema-registry*
- *postgres:15.1*
- *dpage/pgadmin4*

Table des matières

Ateliers 1 : Le cluster kafka

1.1 Premier démarrage et logs

Visualiser le fichier ***docker-compose.yml***

Il permet de démarrer un cluster 3 nœuds ainsi qu'un container RedpandaConsole pour l'administration du cluster

Démarrer cette stack avec :

```
docker compose up -d
```

Observer les logs de démarrages d'un nœud

```
docker logs -f kafka-0
```

1.2 Utilisation des utilitaires

Ouvrir un bash sur un des nœuds

```
docker exec -it kafka-0 bash
```

Créer un topic testing avec 5 partitions et 2 répliques :

```
kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 2 –partitions 5 --topic testing
```

Lister les topics du cluster

Démarrer un producteur de message

```
kafka-console-producer.sh --broker-list localhost:9092 --topic testing --property "parse.key=true" --property "key.separator=:"
```

Saisir quelques messages

Accéder à la description détaillée du topic (commande describe)

Consommer les messages depuis le début

```
kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic testing --from-beginning
```

Dans une autre fenêtre, lister les groupes de consommateurs et accéder au détail du groupe de consommateur en lecture sur le topic testing

1.3 Redpanda Console

Parcourir l'UI de Redpanda Console : http://localhost:9090

Ateliers 2 : Apache Kafka APIS

2.1 Producer API

2.1.1 Découverte du projet

Reprendre le projet Maven ***$TP_DATA/6_KafkaAPIs/6.1_Producer/KafkaProducer***

Le projet est composé de :

- Une classe principale ***KafkaProducerApplication*** qui prend en arguments :
  - ***nbThreads*** : Un nombre de threads
  - ***nbMessages*** : Un nombre de messages
  - ***sleep*** : Un temps de pause
  - ***sendMode*** : Le mode d'envoi : 0 pour *Fire_And_Forget*, 1 pour SYNCHRONE, 2 pour ASYNCHRONE

    L'application instancie <*nbThreads*> de *KafkaProducerThread* et leur demande de s'exécuter.
    Quand toutes les threads sont terminées, elle affiche le temps d'exécution
- Une classe ***KafkaProducerThread*** qui une fois instanciée envoie *nbMessages* tout les temps de pause selon un des 3 modes d'envoi.
  Les messages sont constitués
  - d'une clé au format String: l'id du coursier
    La clé sera sérialisée via la classe
    org.apache.kafka.common.serialization.StringSerializer
  - d'une valeur : Le coursier encapsulant sa Position
    La valeur sera sérialisée grâce à la classe ***org.formation.JsonSerializer***
- Le package ***model*** contient les classes modélisant les données
  - ***Position*** : Une position en latitude, longitude
  - ***Coursier*** : Un coursier associé à une position
  - ***SendMode*** : Une énumération des modes d'envoi

2.1.2 Code du producer

La classe ***KafkaProducerThread*** est à compléter

- Dans la méthode ***_initProducer()***
  Initialiser un *KafkaProducer<String,Coursier> avec les propriétés de configuration suivante :*
  - ***ProducerConfig.BOOTSTRAP_SERVERS_CONFIG***
  - ***ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG***
  - ***ProducerConfig.VALUE_SERIALIZER_CLASS_CONFI***G
- Dans la boucle de la méthode *run()*,
  Construire un ***ProducerRecord*** à chaque itération
- Implémenter les 3 méthodes d'envoi
  - Dans la méthode ***fire_and_forget()***, afficher sur la console le *ProducerRecord* envoyé

- Dans l’envoi synchrone, méthode ***synchronous()*** afficher sur la console la réponse du broker
- Pour l’envoi asynchrone, méthode ***asynchronous()*** créer une classe de callback et y afficher sur la console la réponse du broker

Tester les 3 modes d’envoi

2.2 : Consumer API

2.2.1 Mise en place base Postgres

Démarrer un serveur postgres et sa console d’administration

```
docker compose -f postgres-docker-compose.yml up -d
```

Accéder à ***localhost:81*** et se logger avec ***admin@admin.com/admin***

Enregistre un serveur avec comme paramètres de connexion :

- host : ***consumer-postgresql***
- user : ***postgres***
- password : ***postgres***

Créer une base consumer et y exécuter le script de création de table fourni : ***create-table.sql***

2.2.2 Implémentation

Reprendre le projet Maven ***$TP_DATA/2_KafkaAPIs/2.2_Consumer/KafkaConsumer***

Le projet est composé de :

- Une classe principale ***KafkaConsumerApplication*** qui prend en arguments :
  - ***nbThreads*** : Un nombre de threads

    L’application instancie *<nbThreads> KafkaConsumerThread* et leur demande de s’exécuter. Le programme s’arrête au bout d’un certains temps.
- Une classe ***KafkaConsumerThread*** qui une fois instanciée poll le topic position avec un timeout de 100ms
  Il traite les messages 1 à 1

- Le package ***model*** contient les classes modélisant les données
  - ***Position*** : Une position en latitude, longitude
  - ***Coursier*** : Un coursier associé à une position

2.2.3 Code du consommateur

Compléter la boucle de réception des messages

Pour cela, vous devez

- Initialiser un *KafkaConsumer avec les bonnes propriétés :*
  - *ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG*
  - *ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG*
  - *ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG*
  - *ConsumerConfig.GROUP_ID_CONFIG*
  - *ConsumerConfig.AUTO_OFFSET_RESET_CONFIG* à la valeur ***earliest*** pour recommencer depuis le début
- Implémenter la boucle de réception :
  - simuler un traitement de 10 ms pour le message
  - Afficher le nombre de messages reçu pour un poll et le nombre total de messages reçus depuis le démarrage du consommateur.
- Traiter la *WakeupException*
  - Afficher le nombre de messages total reçus

Pour tester la réception, vous pouvez utiliser le programme précédent et le lancer afin qu'il exécute de nombreux message :

Par exemple :

```
producer_home$ java -jar target/producer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 10 100000 500 0
```

2.2.4 Implémentation de ConsumerRebalanceListener

Modifier la classe *KafkaConsumerThread* afin qu'elle implémente l'interface ***ConsumerRebalanceListener***

Afficher les information de rééquilibrage sur la console dans les 2 méthodes.

Exemple d'implémentation :

```
@Override
public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        System.out.println("Process " +  ProcessHandle.current().pid()
                           + " Thread " +id + " Revocation of " + partitions);

}

@Override
public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
        System.out.println("Process " +  ProcessHandle.current().pid()
                          + " Thread " +id + " Assignement of " + partitions);
}
```

2.2.5 Tests

Une fois le programme mis au point, effectuer plusieurs tests

Tester qu'aucun message n'est perdu :
Démarrer le programme avec 1 thread arrêter puis redémarrer avec la même configuration

Tester la réaffectation de partitions :
Démarrer avec 2 threads

Puis démarrer un autre processus avec 1 threads

Puis un troisième processus avec 1 thread

Arrêter un des processus précédents

2.3 Schema Registry et Avro

2.3.1 Démarrage de Confuent Registry

Visualiser le fichier ***TPS/2_KafkaAPIS/2.3_Schema/docker-compose.yml*** fourni

Démarrer la stack

Accéder à http://*localhost:8081/subjects*

Vérifier que Redpanda se connecte correctement à *Schema Registry*

Consulter le mode de compatibilité du Schema Registry

2.3.2 Producteur de message

Copier le projet *KafkaProducer* dans ***KafkaProducerAvro***

Dépendances
Reprendre ***TPS/2_KafkaAPIs/2.3_Schema/pom.xml*** dans le nouveau projet

Mise au point schéma
Mettre au point un **schéma Avro : src/main/resources/Coursier.avsc**

```
{
    "type": "record",
    "name": "Coursier",
    "namespace": "org.formation.model",
    "fields": [
    {
        "name": "id",
        "type": "string"
    },
    {
      "name": "position",
      "type": [
            {
            "type":     "record",
            "name": "Position",
            "namespace":  "org.formation.model",
            "fields":  [
            {
                  "name": "latitude",
                  "type":     "double"
            },
            {
                  "name": "longitude",
                  "type": "double"
            }
            ]
            }
      ]
    }
    ]
}
```

Supprimer les classes *Coursier* et *Position* du package ***org.formation.model***

Effectuer un ***mvn compile*** et regarder les classes générées par le plugin Avro

Fixer les erreurs de compilation

Enregistrement du schéma sur le serveur
Dans la classe main, changer le nom du topic en ***position-avro*** et définir l'adresse du serveur :

```
	public static String TOPIC ="avro-position";
	public static String REGISTRY_URL = "http://localhost:9091";
```

Dans une méthode init_registry() enregistrer le schéma
sur le serveur programmatiquement :

```
private static void _initRegistry() throws IOException, RestClientException {

	// avro schema avsc file path.
	String schemaPath = "/Coursier.avsc";
	// subject convention is "<topic-name>-value"
	String subject = TOPIC + "-value";

	InputStream inputStream =
KafkaProducerApplication.class.getResourceAsStream(schemaPath);

	Schema avroSchema = new Schema.Parser().parse(inputStream);

	CachedSchemaRegistryClient client = new
CachedSchemaRegistryClient(REGISTRY_URL, 20);

	client.register(subject, new AvroSchema(avroSchema));

}
```

Appeler cette méthode avant le démarrage des threads

Changement configuration KafkaProducer

Dans *KafkaProducerThread , changer la configuration du Producer* :

- `ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG` = **io.confluent.kafka.serializers.KafkaAvroSerializer**
- Qu'elle renseigne la clé **AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG**

Modifier également le nom du *topic* à `KafkaProducerApplication.TOPIC`

Test et vérifications
Tester la production de message.

Accéder à http://localhost:8081/subjects

Puis à http://localhost:8081/schemas/

Vérifier que ***redpanda*** puisse lire les messages.

2.3.3 Consommateur de message

Copier le projet *KafkaConsumer* dans ***KafkaConsumerAvro***

Dépendances
Reprendre le même ***pom.xml*** que dans le projet précédent en changeant l'artifact-id en ***consumer-avro***

Classes du modèle
Ne plus utiliser les classes de modèle mais la classe d'Avro ***GenericRecord***
Supprimer les classes du package ***org.formation.model***
Utiliser la classe générique à la place de la classe Coursier

Configuration *KafkaConsumer*:

- Le désérialiseur de la valeur à : ***"io.confluent.kafka.serializers.KafkaAvroDeserializer"***
- La propriété ***AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG*** à l'adresse du serveur

Ne pas oublier de modifier l'id du groupe également.

Changer le nom du topic et consommer les messages de l'atelier précédent

2.3.4 Evolution du schéma compatible

Mettre à jour le schéma en ajoutant les champs optionnels : ***firstName*** dans la structure ***Coursier***

```
{
      "name": "first_name",
      "type": "string",
      "default": "undefined"
},
```

Fixer les problèmes de compilation

Relancer le programme de production et visualiser la nouvelle version du schéma dans le registre

Consommer les messages sans modifications du programme consommateur

2.3.5 Evolution du schéma incompatible

Mettre à jour le schéma en ajoutant un champs obligatoire : ***vehicle_id*** dans la structure ***Coursier***

```
      {
            "name": "vehicle_id",
            "type": "int"
      },
```

Fixer les problèmes de compilation

Relancer le programme de production et visualiser l'exception au moment de l'enregistrement du nouveau schéma.

Visualiser les nouveaux messages publiés dans le *topic*

2.4 : Initiation à KafkaStream

Objectifs :
- Avoir un premier contact avec une application KafkaStream

Description :
Nous voulons écrire une application qui prend en entrée le topic ***position*** et filtre les positions qui sont supérieures à une certaine latitude.

2.4.1 Reprise du projet

Reprendre le projet de ***TPS/2_KafkaAPIs/2.4_Streams/KafkaStream***

Visualiser les dépendances

Le projet est constituée d'une classe Main typique d'une application *KafkaStream*
- Elle définit les propriétés de configuration :
  - boostrap-servers
  - id de l'application
  - Sérialiseurs/Désérialiseurs sous forme de *Serde*
- Elle définit la topologie de processeur
- Elle démarre l'application en s'étant donné un moyen de l'arrêter

2.4.2 Topologie de processeurs

Ajouter au KStream ***coursierStream*** construit à partir du topic *position* :
- Un processeur qui arrondit la latitude et la longitude à une valeur entière
- Redéfinit la clé de l'évènement comme étant la position du Coursier
- Transforme la valeur de l'événement en juste l'identifiant du coursier
- Sépare en 2 flux :
  - 1 flux Nord : Latitude > 45
  - 1 flux Sud : Latitude ≤ 45
- Déverser les 2 flux dans 2 topics Kafka ***position-nord*** et ***position-sud***

Ateliers 3. Production de messages avec Spring

3.1 Auto-configuration

Récupérer le projet ***TPS/3_SpringKafka/3.1_Templates/OrderService***

Démarrer le projet et accéder à son API : *http://localhost:8180/swagger-ui.html*

Vous pouvez également voir les tables de la BD à : *http://localhost:8180/h2-console*

Ajouter ensuite les dépendances Kafka et la configuration minimale pour déclarer notre cluster Kafka

```
spring:
  kafka:
    bootstrap-servers:
      - localhost:19092, localhost:19093, localhost:19094
```

Visualiser les beans créés par l'auto-configuration et visualiser les propriétés par défaut du *ProducerFactory*

3.2 Implémentation Pattern microservices

Nous voulons implémenter 2 patterns classiques des architecture micro-services

- ***Domain Event Pattern*** *: A chaque fois qu'une opération de mise à jour est effectuée sur l'agrégat Order. Un évènement est publié sur un topic*
- ***Transactional Outbox Pattern*** *: Afin de garantir qu'un message Kafka sera envoyé à chaque mise à jour. Nous commençon par stocker un évènement dans une table de base de données associé à la transaction BD mettant à jour l'agrégat. Une tâche de fond lit la table des évènements et envoie le message vers kafka*

3.2.1 Stockage événement

Récupérer le package ***org.formation.event*** qui contient les classes suivantes :

- ***OrderEventType*** : Enumération typant l'évènement sur l'agrégat Order
- ***OrderEvent*** : Classe Entité persistante encapsulant :
  - Un ID
  - Un type d'évènement
  - L'id de l'agrégat Order
  - Une charge utile contenant les données de l'évènement
- ***OrderEventRepository*** : Permettant de stocker l'évènement dans la base

Sauvegarde de l'événement lors de la création de commande :
Modifier la méthode *createOrder* de *OrderService* pour que dans la même transaction la commande et l'événement associé soit créé.

Par exemple, vous ajouterez les lignes suivantes :

```
Ordervent event = OrderEvent.builder().type(OrderEventType.ORDER_CREATED)
                                .orderId(order.getId())
                                .payload(mapper.writeValueAsString(order))
                                .build();
eventRepository.save(event);
```

3.2.2 Tâche récurrente d'envoi de message

Ajouter l'annotation ***@EnableScheduling*** sur la classe principale du projet

Créer la classe ***org.formation.service.EventService.***
La classe est responsable de périodiquement traiter les enregistrements présents dans la table OrderEvent.
Pour chaque événement trouvé, elle s'appuie sur le bean ***KafkaTemplate*** pour envoyer l'événement vers Kafka.
Si l'envoi est réussi il supprime l'événement dans la table

Utiliser le sérialiseur ***org.springframework.kafka.support.serializer.JsonSerializer*** pour sérialiser *OrderEvent* ainsi que ***org.apache.kafka.common.serialization.LongSerializer***
pour la clé

3.3 KafkaRoutingTemplate

Récupération du projet et description

Récupérer le projet ***TPS/3_SpringKafka/3.3_RoutingTemplate/DeliveryService***

Vérifier son démarrage et son API Rest :
http://localhost:8083/swagger-ui/index.html

Le projet désire envoyer des messages vers 2 topics différents:

- ***livraisons*** : Envoyer les statuts des livraisons
- ***coursiers*** : Envoyer les position des coursiers

Nous voulons que le *ProducerKafka* ne soit pas configuré de la même façon dans les 2 cas :

- Vers livraisons, nous voulons favoriser la sûreté de fonctionnement
  - acks=all
- Vers coursiers, nous voulons favoriser la latence :
  - acks=0
  - LINGER_MS_CONFIG = 0
  - RETRIES_CONFIG = 0
  - MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION = 5

3.3.1 Configuration des topics

Créer une classe ***org.formation.KafkaConfig***

Définir 2 beans de type ***NewTopic*** créant les topics livraisons et coursiers

```
@Bean
NewTopic coursierTopic() {
   return TopicBuilder.name(coursierChannel).partitions(10).replicas(2).build();
}

@Bean
NewTopic livraisonTopic() {
   return TopicBuilder.name(livraisonChannel).partitions(3).replicas(3)
         .config(TopicConfig.MIN_IN_SYNC_REPLICAS_CONFIG, String.valueOf(3)).build();
}
```

3.3.2 Envoi de messages

Dans le bean ***org.formation.service.CoursierService***, nous voulons implémenter une production de type *fire and forget*
Injecter :

- Une valeur de configuration représentant le nom du topic
- Une classe ***RoutingKafkaTemplate***

Rajouter l'envoi de la position dans la méthode ***moveCoursier()*** :

```
routingTemplate.send(coursierChannel, coursier.getId(), coursier.getPosition());
```

Dans le bean ***org.formation.service.LivraisonService*** implémenter un envoi asynchrone avec traitement d'erreur
Injecter :

- Une valeur de configuration représentant le nom du topic
- Une classe ***RoutingKafkaTemplate***

Ajouter le code d'envoi similaire à :

```
routingTemplate.send(livraisonChannel, livraison.getId(), livraison).whenComplete((r, e) -> {
   if (e != null) {
         log.info("ERROR when sending : " + e.toString());
      ret.setStatus(LivraisonStatus.FAILED);
      livraisonRepository.save(livraison);
   } else {
      log.info("Message sent to Livraison channel offset : " + r.getRecordMetadata().offset());
   }
});;
```

3.3.3 Tests

Utiliser le script JMeter DeliveryService.jmx pour provoquer l'envoi de message.

3.4 ReplyingKafkaTemplate

Description :
Le service *OrderService* envoie une demande de paiement et récupère une réponse pour mettre à jour la classe de domaine ***PaymentInfo*** avec le ***transactionId***

3.4.1 Reprise du projet PaymentService

Reprendre le projet ***TPS/3_SpringProduction/3.4_ReplyingTemplate/PaymentService***

Vérifier son démarrage et son API Rest :
http://localhost:8182/swagger-ui/index.html

Le projet traite des ordres de paiements et génère un ID de transaction. (voir ***org.formation.service.PaymentService***)

3.4.2 Configuration Kafka de OrderService

Création de 2 topics : 1 pour les requêtes, 1 pour les réponses

```
@Bean
NewTopic requestTopic() {
   return TopicBuilder.name(requestPaymentChannel).partitions(3).replicas(3).build();
}

@Bean
NewTopic responseTopic() {
   return TopicBuilder.name(responsePaymentChannel).partitions(3).replicas(3).build();
}
```

Dans la classe KafkaConfig, configurer des beans *ConcurrentMessageListenerContainer*, *ReplyingKafkaTemplate et KafkaTemplate (pour l'envoi dans le channel Orders)*

```
@Bean
public ConcurrentMessageListenerContainer<String, String> repliesContainer(
      ConcurrentKafkaListenerContainerFactory<String, String> containerFactory) {
   ConcurrentMessageListenerContainer<String, String> repliesContainer =
containerFactory.createContainer(responsePaymentChannel);
   repliesContainer.getContainerProperties().setGroupId("order-service");
   repliesContainer.setAutoStartup(false);
   return repliesContainer;
}
@Bean
public ReplyingKafkaTemplate<String, String, String> replyingKafkaTemplate(
      ProducerFactory<String, String> pf,
      ConcurrentMessageListenerContainer<String, String> repliesContainer) {
   return new ReplyingKafkaTemplate<>(pf, repliesContainer);
}
@Bean
public KafkaTemplate<Long, OrderEvent> kafkaTemplate(
                  DefaultKafkaProducerFactory<Long, OrderEvent> factory) {
   return new KafkaTemplate<Long, OrderEvent>(factory);
}
```

Modifier l'injection dans *EventService* pour utiliser un injection par nom

```
@Autowired
private OrderEventRepository eventRepository;
@Resource
private KafkaTemplate<Long, OrderEvent> kafkaTemplate;
```

3.4.3 Request/Response dans OrderService

Dans ***org.formation.service.OrderService***, après avoir sauvegardé la nouvelle commande on effectue une requête de paiement via le topic « ***payments-in*** »

On traite ensuite la réponse comme suit :

- Si OK, mise à jour de l'entité *Order* avec le statut *OrderStatus.APPROVED* et le *transactionId* renvoyé. Stocker également un *OrderEvent* correspondant
- Si NOK mise à jour de *Order* avec le statut *OrderStatus.REJECTED* et stocker un *OrderEvent* encapsulant l'exception

Une implémentation possible :

```
order.getPaymentInformation().setAmount(order.getTotal());
ProducerRecord<Long, PaymentInformation> record = new ProducerRecord<Long,
PaymentInformation>(requestPaymentChannel, order.getId(), order.getPaymentInformation());
replyingKafkaTemplate.sendAndReceive(record).whenComplete((reply, ex) -> {
        OrderEvent event2 = null;
        if (ex != null) {
                log.info("Error in Request/Response");
                ret.setStatus(OrderStatus.REJECTED);
                event2 =
OrderEvent.builder().type(OrderEventType.ORDER_CANCELLED).orderId(ret.getId()).payload(ex.
getMessage()).build();
        } else {
                log.info("GET a transactionId " + reply.value());
                ret.getPaymentInformation().setTransactionId(reply.value());
                ret.setStatus(OrderStatus.APPROVED);
                try {
                        event2 =
OrderEvent.builder().type(OrderEventType.ORDER_PAID).orderId(ret.getId()).payload(mapper.w
riteValueAsString(ret)).build();
                } catch (JsonProcessingException e) {
                        throw new RuntimeException(e);
                }

        }
        orderRepository.save(ret);
        eventRepository.save(event2);
```

3.4.4 Listener dans PaymentService

Configurer Kafka via ***application.yml***

```
spring:
  kafka:
    bootstrap-servers:
    - localhost:19092, localhost:19093, localhost:19094
    producer:
      value-serializer: org.apache.kafka.common.serialization.StringSerializer
      key-serializer: org.apache.kafka.common.serialization.LongSerializer
    consumer:
      value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer
```

```
    key-deserializer: org.apache.kafka.common.serialization.LongDeserializer
    properties:
      spring.json.trusted.packages: '*'
```

Dans *PaymentService*, créer un bean service ***org.formation.service.EventHandler***

Annoter une méthode avec ***@KafkaListener*** et ***@SendTo*** qui traite les demandes de paiement.

```
@Autowired
PaymentService paymentService;

@KafkaListener(id="payment-service", topics = "payments-in")
@SendTo
String processPayment(PaymentInformation paymentInformation ) throws PaymentException {
        return paymentService.processPayment(
                    paymentInformation.getFromAccount(),
                    paymentInformation.getToAccount(),
                    paymentInformation.getAmount());
}
```

Tests

Pour tester , utiliser le script JMeter CreateOrder.jmx

4. Consommation de message

Objectifs :

- Distinguer les traitements par batch des traitements individuels de messages
- Voir 2 gestions des commits

4.1 Consommation par record et commit manuel

4.1.1 Reprise du projet TicketService

Reprendre le projet ***TPS/4_SpringConsommation/4.1_Record/TicketService***

Vérifier son démarrage et son API Rest :
http://localhost:8183/swagger-ui/index.html

Le projet traite les commandes sous forme de Ticket

4.1.2 Configuration Kafka

Configurer Kafka :

Le consumer avec :

- Auto-offset-reset à earliest
- key-deserialiser : Long
- Value-deserialiser : Json
- La propriété ***spring.json.trusted.packages*** à ***org.formation.event***

Le listener avec :

- AckMode à ***manual-immediate***
- concurrency : **3**

4.1.3 Implémentation de la consommation

Dans la classe ***org.formation.service.Eventhandler***, créer une méthode annotée par *@KafkaListener*

Si le type de l'événement est ORDER_PAID, créer un Ticket et committer l'offset

```
public void handleOrderEvent(@Payload OrderEvent orderEvent, Acknowledgment
acknowledgment) throws JsonMappingException, JsonProcessingException {
   switch (orderEvent.getType() ) {
      case "ORDER_PAID":
         log.info("Oder paid creating Ticket");
         OrderDto orderDto = mapper.readValue(orderEvent.getPayload(), OrderDto.class);
         ticketService.createTicket(orderDto);
         acknowledgment.acknowledge();
         break;
```

```
    }
}
```

4.1.4 Test

Recréer le topic Order avec un nombre de partition est égal à 3.

Utiliser le script JMeter CreateOrder.jmx pour visualiser la consommation, surveiller le lag du consommateur

4.2 Traitement par lot et minimisation des commits

Description :

Les coursiers envoient très régulièrement leur position. Non seulement la perte de messages n'est pas important mais nous voulons également minimiser les écritures en base

Le projet Delivery reçoit de grand batch de messages pouvant contenir plusieurs messages pour le même coursier. Nous allons enregistrer en base seulement la dernière position.

D'autre part, nous voulons adapté le degré de parallélisme du traitement au nombre de partitions du topic

4.2.1 Reprise du nouveau projet

Reprendre le projet ***TPS/4_SpringConsommation/4.2_Batch/PositionService*** qui offre un endpoint http permettant à un coursier d'indiquer sa position

Le démarrer et tester : http://localhost:8184/swagger-ui.html

4.2.2 Configuration via application.yml

La configuration du listener positionne le mode batch, un commit automatique tous les 500 enregistrements et temps d'attente entre les appels à poll

```yaml
spring:
  kafka:
    bootstrap-servers:
    - localhost:19092, localhost:19093,localhost:19094
    consumer:
      group-id: delivery-service
      key-deserializer: org.apache.kafka.common.serialization.LongDeserializer
      value-deserializer:
org.springframework.kafka.support.serializer.JsonDeserializer
      properties:
        spring.json.trusted.packages: '*'
    listener:
      type: batch
      concurrency: 10
      client-id: position-consumer
      ack-mode: count
      ack-count: 500
      idle-between-polls: 100
```

4.2.3 Implémentation de la réception

Créer une classe ***org.formation.service.EventHandler*** et implémenter une méthode recevant un batch de données sous forme de classe *Position* et de clés de coursier.

Pour chaque Coursier, récupérer sa dernière position et la sauvegarder en base.

```
@KafkaListener(topics = "${app.coursier-channel}" )
public void handleCoursierPosition(List<Position> positions,
		@Header(KafkaHeaders.RECEIVED_KEY) List<Long> coursierIds) {
	Map<Long,Position> map = new HashMap<>();
	for (int i=0; i< positions.size(); i++) {
		map.put(coursierIds.get(i), positions.get(i));
	}
	log.info(positions.size() + " positions received  Pour " + map.size() + " coursiers");

	map.forEach((id,position) -> {
		Coursier coursier =
coursierRepository.findById(id).orElse(Coursier.builder().id(id).build());
		coursier.setPosition(position);
		coursierRepository.save(coursier);
	});
}
```

4.2.4 Tests

Utiliser le script JMeter ***TPS/8_SpringKafka/8.2_Listener/Position.jmx*** fourni pour générer beaucoup de messages

Visualiser la taille du batch (nombre de données reçues par la méthode)

Faire varier ***idle-between-polls***

Arrêter *DeliveryService* afin qu'il prenne du lag, puis le redémarrer

Visualiser les commits d'offsets par exemple en regardant les messages du topic ***__consumer_offsets***

Essayer de trouver une configuration qui stabilise le lag du consommateur

Ateliers 5 Transaction et Exactly Once

Objectifs :

- Envoyer 2 messages en une seule transaction
- Appliquer la sémantique Exactly Once à l'interaction *Order → PaymentService → Order*

5.1 Producteur transactionnel

Dans le projet OrderService, lors d'une création d'une commande nous voulons envoyer 2 messages en une seule transaction :

- Un message sur le topic orders qui notifie le changement de statut d'une commande
- Un message de déclenchement de Paiement vers PaymentService

Les messages sont envoyés par le service EventService

5.1.1 Transactions bases de données

Dans ***org.formation.service.OrderService***, la méthode **createOrder()** est également transactionnelle mais utilise que le gestionnaire de transaction base de données, l'annoter avec ***@Transactional("transactionManager")***
La méthode consiste à écrire dans la table Order et OrderEvent

5.1.2 Transaction Kafka

Le code présent dans ***org.formation.service.OrderService*** est donc à déplacer dans ***EventService***.

Annoter la méthode avec ***@Transactional("kafkaTransactionManager")***

```
@Scheduled(fixedDelay = 10l, timeUnit = TimeUnit.SECONDS)
@Transactional("kafkaTransactionManager")
public void relayEvents() {
   eventRepository.findAll()
         .forEach(event -> {
            log.info("Sending event"+event);
            kafkaTemplate.send(ORDER_CHANNEL, event.getOrderId(), event);
            if ( event.getType().equals(OrderEventType.ORDER_CREATED) ) {
                requestPayment(event.getPayload());
            }
            eventRepository.delete(event);
         });
}
```

Dans la configuration, spécifier la propriété :

```
spring:
  kafka:
      transactionIdPrefix: "order-tx-"
```

Pour bien voir ce qui se passe augmenter le niveau de log des gestionnaires de transaction :

```yaml
logging:
  level:
    org.springframework.transaction: trace
    org.springframework.kafka.transaction: debug
```

Effectuer une requête POST et visualiser les traces

5.2 Consommation et envoi d'une réponse transactionnel

Le service ***PaymentService*** ne doit lire que les messages committés, il renvoie une réponse que l'on peut également inclure dans une transaction.

La configuration correspondante est donc :

```yaml
spring:
  kafka:
    producer:
      transactionIdPrefix: "payment-tx-"
    consumer:
      isolation-level: READ_COMMITTED
```

Ajouter également

```yaml
logging:
  level:
    org.springframework.transaction: trace
    org.springframework.kafka.transaction: debug
```

Effectuer une requête POST sur *OrderService* et visualiser les traces de *PaymentService*

5.3 Test via JMeter

Utiliser le fichier JMeter ***TPS/5_Transactions/CreateOrderTransacation.jmx*** pour tester des transactions validées et non validées

Ateliers 6 : Sérialisation

Objectifs : Pouvoir envoyer et consommer plusieurs types de message sur un même topic en utilisant ***DelegatingSerializer*** et ***DelegatingDeserializer***

6.1 Production de messages

Reprendre le projet ***PositionService***.

Désormais, nous voulons envoyer sur le topic coursier soit une *Position* soit une simple *String* indiquant une action du coursier (STOP, START, PAUSE, RESUME, etc.)

Nous effectuons une configuration programmatique via une classe *KafkaConfig*

Reprendre les informations précédemment spécifiées dans *application.yml* et ajouter les informations de configuration propre au *DelegatingSerializer*

```
@Configuration
public class KafkaConfig {

	@Bean
	public ProducerFactory<Long, Object> producerFactory() {
	   return new DefaultKafkaProducerFactory<>(producerConfigs());
	}

	@Bean
	public Map<String, Object> producerConfigs() {
	   Map<String, Object> props = new HashMap<>();
	   props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:19092,
localhost:19093,localhost:19094");
	   props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, LongSerializer.class);
	   props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, DelegatingSerializer.class);
	   props.put(DelegatingSerializer.VALUE_SERIALIZATION_SELECTOR_CONFIG,
	            "commande:org.apache.kafka.common.serialization.StringSerializer,
position:org.springframework.kafka.support.serializer.JsonSerializer");

	   props.put(ProducerConfig.RETRIES_CONFIG, 5);
	   props.put(ProducerConfig.ACKS_CONFIG, "all");
	   return props;
	}

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

Désormais lors de l'envoi du message, il est nécessaire de positionner l'entête ***DelegatingSerializer.VALUE_SERIALIZATION_SELECTOR*** avec la clé correspondante

Le endpoint permettant l'envoi d'action est alors le suivant :

```
@PatchMapping("{id}/{commande}")
Mono<Void> stop(@PathVariable("id") long id, @PathVariable("commande") String commande) {

	List <Header> headers = new ArrayList<>();
	headers.add(new RecordHeader(DelegatingSerializer.VALUE_SERIALIZATION_SELECTOR,
	"commande".getBytes()));

	ProducerRecord<Long, Object> record = new ProducerRecord<Long, Object>(coursierChannel, null, id,
commande,headers);

	kafkaTemplate.send(record);

	return Mono.empty();
```

```
}
```

Modifier de la même façon le endpoint permettant d'envoyer les positions

6.2 Consommation de messages

Reprendre le projet ***DeliveryService***

Dans ce projet nous effectuons la configuration du *DelegationDeserializer* par *application.yml* :

Attention, le type batch n'est pas supporté dans ce cas :

```
spring:
  kafka:
    consumer:
      value-deserializer:
org.springframework.kafka.support.serializer.DelegatingDeserializer
      properties:
        spring.kafka.serialization.selector.config:
"commande:org.apache.kafka.common.serialization.StringDeserializer,
position:org.springframework.kafka.support.serializer.JsonDeserializer"
    listener:
#      type: batch
```

Modifier la classe ***org.formation.service.EventHandler*** afin qu'elle propose 2 méthodes de réception : une pour les objets Position, l'autre pour les objets String

Déplacer l'annotation @KafkaListener sur la classe et marquer les méthodes par ***@KafkaHandler***

```
@Service
@Log
@KafkaListener(topics = "${app.coursier-channel}" )
public class EventHandler {

	@Autowired
	CoursierRepository coursierRepository;

	@KafkaHandler
	public void handleCoursierPosition(Position position,
			@Header(KafkaHeaders.RECEIVED_KEY) List<Long> coursierIds) {
		log.info("Receiving position");
	}

	@KafkaHandler
	public void handleCommande(String commande,
			@Header(KafkaHeaders.RECEIVED_KEY) Long coursierId) {
		log.info("Receiving commande");
	}
}
```

6.3 Test

Tester les 2 endpoints de *PositionService* avec le script JMeter ***PositionCommande.jmx***

Ateliers 7 : Gestion des Exceptions

Objectifs : Savoir mettre en place les stratégies classiques de traitement d'exception

7.1 Exception métier :

Reprendre le projet Delivery-service

Modifier le code métier afin qu'il lance une exception tous les x messages

```
@KafkaHandler
public void handleCommande(String commande,
                @Header(KafkaHeaders.RECEIVED_KEY) Long coursierId,
                @Header(KafkaHeaders.OFFSET) int offset) {
        log.info("Receiving commande offset is " + offset);

        if ( offset%10 == 0 ) {
                throw new RuntimeException("Boom");
        } else {
                process++;
        }

}
```

Utiliser le script JMeter fourni ***TPS/7_Exceptions/7.1_Métier/Commandes.jmx*** pour générer un certain nombre de messages de type commande

7.1.1 Configuration par défaut

Visualiser la trace générée par le consommateur et s'assurer que le consommateur rattrape son lag

7.1.2 Configuration d'un gestionnaire par défaut

Modifier le code du listener afin de visualiser les différents cas :

Configurer un gestionnaire d'exception qui essaie 4 tentatives toutes les secondes avant de transférer dans un topic DeadLetter

```
@Bean
public CommonErrorHandler errorHandler(@Qualifier("kafkaProducerFactory")
ProducerFactory<Long, Object> pf) {
  return new DefaultErrorHandler(
    new DeadLetterPublishingRecoverer(new KafkaTemplate<>(pf)), new FixedBackOff(1000L,
4));
}
```

Relancer le script et visualiser les traces de *DeliveryService*

Visualiser les messages dans la DeadLetter-Topic

7.2 Erreur de sérialisation :

7.2.1 Comportement par défaut

Récupérer le service ***TPS/7_Exceptions/7.2_Désérialisation/PaymentBadService*** et visualiser son code et les sérialiseurs utilisés

Démarrer également ***PaymentService***

Envoyer un mauvais message via l'url de *PaymentBadService*

POST http://localhost:8186/api/bad

Visualiser la console de *PaymentService*

7.2.2 Configuration ErrorHandlingDeserializer dans PaymentService

Configurer un ***ErrorHandlingDeserializer*** pour la valeur

```
spring:
  kafka:
    consumer:
      value-deserializer:
org.springframework.kafka.support.serializer.ErrorHandlingDeserializer
      properties:
        spring.deserializer.value.delegate.class:
org.springframework.kafka.support.serializer.JsonDeserializer
```

Redémarrer *PaymentService* et visualiser la console

7.2.3 Ajout d'une Valeur de Fallback

Crée une classe ***org.formation.domain.BadPaymentInformation*** comme suit :

```
@EqualsAndHashCode(callSuper = true)
@Data
public class BadPaymentInformation extends PaymentInformation {

        private final FailedDeserializationInfo failedDeserializationInfo;

          public BadPaymentInformation(FailedDeserializationInfo failedDeserializationInfo) {
           this.failedDeserializationInfo = failedDeserializationInfo;
          }


}
```

Ainsi que le fournisseur de la valeur de fallback
org.formation.service.FailPaymentInformationProvider

```
public class FailPaymentInformationProvider implements Function<FailedDeserializationInfo,
PaymentInformation> {

        @Override
        public PaymentInformation apply(FailedDeserializationInfo t) {
                return new BadPaymentInformation(t);
        }

}
```

Configurer ***ErrorHandlingDeserializer*** pour qu'il utilise le fournisseur de fallback

```
spring:
  kafka:
    consumer:
```

```
        spring.deserializer.value.function:
org.formation.service.FailPaymentInformationProvider
```

Modifier le code de *EventHandler* afin de logger les mauvaises valeurs et d'éviter les exceptions

```
@KafkaListener(id="payment-service", topics = "payments-in")
@SendTo
@Transactional("kafkaTransactionManager")
String processPayment(PaymentInformation paymentInformation ) throws PaymentException {
        log.info("Receiving Payment Request with : " + paymentInformation);
        if ( paymentInformation.getFromAccount() == null ||
paymentInformation.getToAccount() == null ) {
                log.info("SKIPPING PROCESS of BadPaymentInfo ");
                return "NOK";
        }
        return paymentService.processPayment(paymentInformation.getFromAccount(),
paymentInformation.getToAccount(), paymentInformation.getAmount());
}
```

Ré-effectuer le test précédent

Atelier 8 : Tests

8.1 Test de la production

Dans le projet PositionService utiliser un serveur embarqué et envoyer un message via la classe Service.

Utiliser KafkaTestUtil pour vérifier que le message a été envoyé

8.2 Test de la consommation

Dans le projet DeliveryService, utiliser un serveur embarqué et configurer un KafkaTemplate avec le serveur embarqué associé.

Tester la consommation de messages

8.3 TestContainer

Effectuer le même test avec un container démarré via TestContainer

Atelier 9: Sécurité

9.1 Séparation des échanges réseaux

Le fichier ***docker-compose*** sépare déjà sur des ports différents la communication contrôleurs, brokers et client externe .

Visualisez la configuration bitnami des listeners

9.2 Accès via SSL au cluster

Redémarrer le cluster en utilisant la nouvelle version de docker-compose, le port EXTERNAL utilise dorénavant SSL

Récupérer le répertoire SSL qui contient un keystore et un trustore self-signed pour localhost

Modifier docker-compose.yml afin que le répertoire ssl/mount soit correctement monté sur chaque broker.

Démarrer le cluster et vérifier le certificat produit par le serveur :

```
openssl s_client -debug -connect localhost:9192 -tls1_2
```

Modifier les propriétés du client SpringProducer afin qu'il utilise le protocle SSL pour communiquer

```
spring:
  kafka:
    bootstrap-servers: localhost:19092
    ssl:
      trust-store-location: file:///home/dthibau/Formations/Kafka/github/kafka-solutions/ssl/mount/kafka.truststore.jks
      trust-store-password: secret
```

Vérifier la production de message

9.3 Authentification avec SASL/PLAIN

9.3.1 Authentification inter-broker

Mettre au point un fichier ***kafka_server_jass.conf*** définissant 2 utilisateurs admin et alice et indiquant que le serveur utilise l'identité admin comme suit :

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

Récupérer la nouvelle version de docker-compose et vérifier le montage de volume vers votre fichier ***kafka_server_jaas.conf***

Redémarrer le cluster et vérifier son bon démarrage

9.3.2 Authentification client

Configurer le projet OrderService pour qu'il s'authentifie avec l'utilisateur ***alice***

Tester l'envoi de message, vérifier la bonne configuration du KafkaProducer et la production de message

9.4 ACL

Visualiser la nouvelle configuration des brokers dans le nouveau ***docker-compose.yml***

Démarrer le cluster.

Vérifier via Redpanda Console que le lien sécurité permet de créer des ACLs

Interdire à Alice la production de message et tester

9.5 Quotas

Se créer un ficier client-admin.properties permettant de se logger avec l'utilisateur admin

Définition d'un quota pour l'utilisateur alice

```
$KAFKA_DIST/bin/kafka-configs.sh --bootstrap-server localhost:9094 --alter --
add-config 'producer_byte_rate=1024,consumer_byte_rate=1024' --entity-type users
--entity-name alice --command-config client-admin-ssl.properties
```

Relancer une application et observer les messages d'erreurs produits

Supprimer les quotas :

```
$KAFKA_DIST/bin/kafka-configs.sh --bootstrap-server localhost:9094 --alter --
delete-config producer_byte_rate --entity-type users --entity-name alice --
command-config client-admin-ssl.properties$KAFKA_DIST/bin/kafka-configs.sh --
bootstrap-server localhost:9094 --alter --
delete-config consumer_byte_rate --entity-type users --entity-name alice --
command-config client-admin-ssl.properties
```