Ateliers
« Spring / Kafka »

Pré-requis :

- Bonne connexion Internet
- Système d'exploitation : Linux / Windows / MacOS
- JDK17+, 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.

Table des matières

Atelier 1 : Rappels Spring

Objectifs de l'atelier :

- Comprendre les avantages du pattern IoC et d'injection de dépendances
- Revoir les principales annotations de Spring
- Mise en place de profils

Description :
Nous voulons développer un service métier capable de lister tous les films d'un réalisateur.
Le service s'appuie sur une couche de persistance définie par une interface

1.1 Mise en place du projet

Récupérer le projet Maven fourni ***1_Spring/MovieApplication***

Visualisez les dépendances Maven, la classe du modèle et l'interface de la couche de persistance

1.2 Mise en place des beans Spring

Bean service
Créer une nouvelle classe ***org.formation.service.MovieService***

L'annoter avec ***@Service***

Implémenter la méthode : `public List<Movie> moviesDirectedBy(String director)` qui s'appuiera sur l'interface *MovieDao*

Classe de Configuration
Créer ensuite ***org.formation.MovieApplication*** et l'annoter avec ***@Configuration***, ***@ComponentScan*** et ***@PropertySource***

Implémentation DAO
Une implémentation de cette interface pour un fichier tabulé respectant un format particulier est fournie (*org.formation.dao.FileDAO*).
Annoter la classe avec le bon stéréotype Spring, injecter la propriété de configuration ***movie.file*** défini dans ***src/main/resources/application.properties***

1.3 Initialisation Contexte et Test

Compléter la classe de test afin d'initialiser Spring et de récupérer le bean ***movieService***
Exécuter ensuite le test et vérifier qu'il passe

1.4 Profils

Nous voulons pouvoir exécuter les tests dans 2 profils distinct « ***file*** » et « ***jdbc*** »

1.4.1 Mise en place de la base de données

Démarrer la base de données avec :
```
docker-compose -f postgres-docker-compose.yml up -d
```

Se connecter sur *pgAdmin* (localhost:81) avec ***admin@admin.com/admin***
déclarer le serveur avec comme propriétés de connexion :
movies-postgresql postgres/postgres
Créer une BD nommée ***movies*** et exécuter le script d'initialisation ***initbd_pg.sql*** fourni

1.4.2 Mise en place des profils

Récupérer les classes JDBC fournies et les annoter correctement.

- ***org.formation.JdbcConfiguration*** : *@Configuration, @Profile* et *@PropertySource*
- ***org.formation.dao.JdbcMovieDao*** : *@Repository* et *@Profile*

1.4.3 Initialisation du contexte et des profils et tests

Réécrire la classe de test en définissant 2 méthodes :

- 1 effectuant le test dans le profile ***file***
- l'autre dans le profil ***jdbc***

Atelier 2 : Spring Boot

2.1 Spring Intializr et Build

Objectifs de l'atelier :

- Utiliser l'assistant de création de projet
- Comprendre les apports fournis par le plugin Maven/Gradle de SpringBoot

Description :
Cet atelier crée une application Web minimale.

Accéder à https://start.spring.io/

Renseigner l'assistant avec les valeurs suivantes :

- Project : **Maven**
- Language : **Java**
- SpringBoot : **Dernière version stable**
- Group : **org.formation**
- artifact : **greetings**
- package : **org.formation.greetings**
- Packaging : **jar**
- Java : **17**
- Dependencies : **Reactive Web** et **Actuator**

Télécharger le projet et le dézipper dans un répertoire de travail

Ajouter dans ***src/main/java/org/formation/greetings*** un fichier nommé ***GreetingsController*** contenant le code suivant :

```
package org.formation.greetings;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingsController {


	@RequestMapping("/hello")
	public String hello(@RequestParam String name) {
		return "Welcome " +name;
	}
}
```

Placer vous dans le répertoire projet et effectuer un build :
`./mvnw clean package`

Vérifier la bonne exécution du build, puis lancer l'application via :
`java -jar target/greetings-0.0.1-SNAPSHOT.jar`

Accéder au contrôleur

Démarrer l'application sous un autre port
`java -jar target/greetings-0.0.1-SNAPSHOT.jar –server.port=8000`

Créer une image OCI avec :
`./mvnw spring-boot :build-image`

Exécuter l'image via :
`docker run -p 8080:8080 greetings:0.0.1-SNAPSHOT`

Modifier ensuite la configuration du plugin springboot (pom.xml) avec le bloc XML suivant

```
<build>
	<plugins>
		<plugin>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-maven-plugin</artifactId>
			<executions>
				<execution>
					<goals>
						<goal>build-info</goal>
					</goals>
				</execution>
			</executions>
		</plugin>
	</plugins>
</build>
```

Dans ***application.properties***, ajouter la ligne suivante :

```
management.endpoints.web.exposure.include=health,info
```

Reconstruire et démarrer l'application, par exemple via
`./mvnw spring-boot:run`

Accéder ensuite aux endpoints suivants :

- http://localhost:8080/actuator
- http://localhost:8080/actuator/health
- http://localhost:8080/actuator/info

2.2 Développer avec Spring Boot

Objectifs de l'atelier :

- Découvrir les fonctionnalités offertes par SpringTools Suite
- Découvrir les fonctionnalités offertes par les starters de développement *DevTools* et *ConfigurationProcessor*
- Gestion des propriétés de configuration applicative et des profils
- Savoir configurer le niveau de trace
- Visualiser le rapport d'auto-configuration SpringBoot

Description :
Cet atelier suppose que l'on utilise Spring Tools Suite version Eclipse. Il utilise le projet précédent

2.2.1 Configuration de démarrage

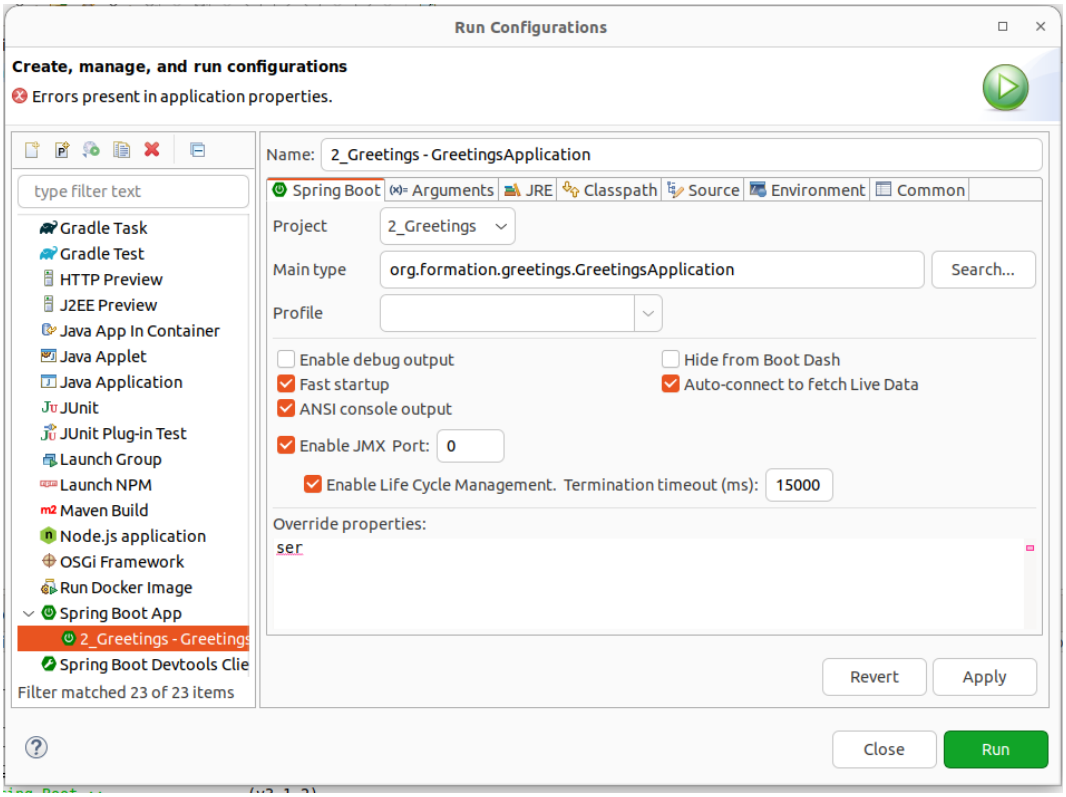
Importer le projet précédent dans SpringToolsSuite :
File → Import → Existing Maven Project

Démarrer le projet dans l'IDE :

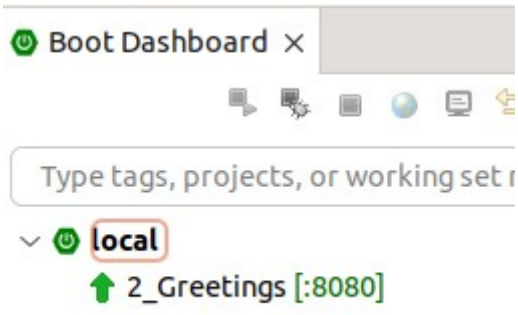
Sélection du projet puis *Run As → Spring Boot App*

Positionner une propriété de configuration dans une configuration de démarrage

Run → Configurations, surcharger la propriété *server.port* à la valeur 8000



Redémarrer l'application via le ***Boot Dashboard***



2.2.2 Starters de développement

Ajouter la dépendance ***devtools*** sur le projet

Sélection du projet, *Spring → Add DevTools*

Redémarrer par le boot dashboard

Éditer une fichier source Java ou *application.properties* et observer le redémarrage

Créer une classe ***org.formation.GreetingsProperties*** contenant le code suivant :

```
@Component
@ConfigurationProperties("hello")
public class GreetingsProperties {

	/**
	 * Greeting message returned by the Hello Rest service.
	 */
	private String greeting = "Welcome ";

	public String getGreeting() {
		return greeting;
	}

	public void setGreeting(String greeting) {
		this.greeting = greeting;
	}
}
```

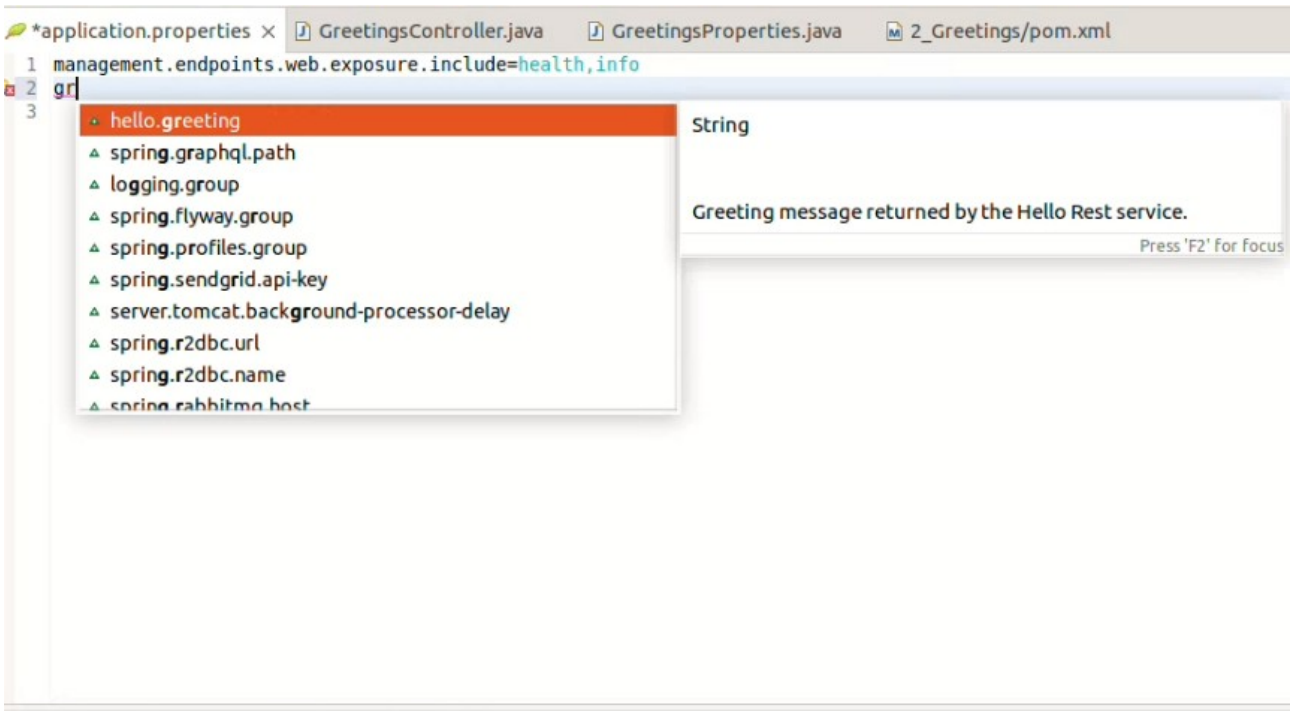
Modifier la classe contrôleur afin que le composant précédent soit injecté et utilisé dans la réponse de la méthode `hello()` :

```
	@RequestMapping("/hello")
	public String hello(@RequestParam String name) {
		return props.getGreeting()+name;
	}
```

Ajouter ensuite la dépendance sur ***`spring-boot-configuration-processor`***, par exemple :

Sélection du projet, *Spring → Add starter*

Tester ensuite la complétion dans l'éditeur du fichier ***application.properties***



2.2.3 Validation des propriétés applicatives et format .yml

Convertir le fichier *application.properties* en *application.yml*

Sélection de *application.properties*, *Convert properties to yaml*

Ajouter ensuite le starter validation

Sélection du projet, *Spring → Add starter*, Choisir IO/Validation

Définir les propriétés applicatives suivantes, ces propriétés sont encapsulées dans la classe de propriétés de configuration *GreetingsProperties* :

- *hello.greeting* (non vide sans valeur par défaut) : La façon de dire *bonjour*
- *hello.styleCase* (Upper ou Lower) : La façon d'écrire le nom
- *hello.position* (0 ou 1) : Le nom est en premier ou en seconde position

Ajouter des contraintes de validation sur les attributs de *GreetingsProperties*

Vérifier :

- Que les contraintes de validation sont effectives
- Que les nouvelles propriétés apparaissent dans la complétion de l'éditeur de *application.yml*

2.2.4 Profils

Éditer *application.yml* et définir un port d'écoute différent le profil ***prod***

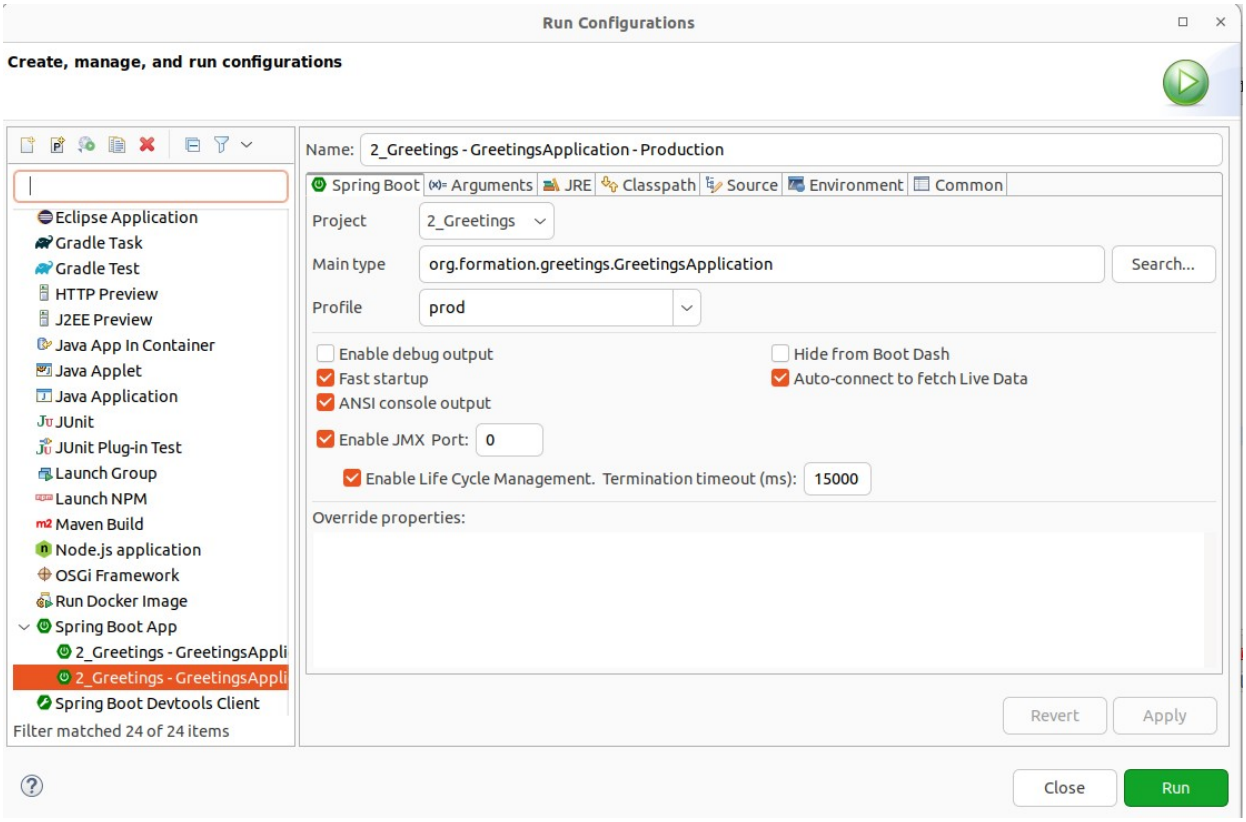
Dupliquer la configuration de démarrage existante et l'editer

Bootstrap View, Sélection de la configuration de démarrage *Duplicate Config*

Puis

Bootstrap View, Sélection de la nouvelle config *Open Config*

Dans cette nouvelle configuration activer le profil ***prod***



Tester le démarrage de cette nouvelle configuration

2.2.5 Mode DEBUG et Configuration des traces

Editer la configuration de démarrage par défaut et cocher ***Enable debug output***

Redémarrer l'application dans cette configuration et visualiser les traces et le rapport d'auto-configuration

Editer application.yml et définir les propriétés ***logging.file.name*** et ***logging.file.path***

```yaml
logging:
  file:
    name: server.log
    path: .
```

Redémarrer et visualiser le fichier généré

Modifier le niveau de trace du logger *org.springframework.boot* à DEBUG après avoir décoché l'option ***Enable debug output***

```yaml
logging:
  level:
    '[org.springframework.boot]': debug
```

Atelier 3 : Restful avec Spring

Objectifs de l'atelier :

- Comprendre les contraintes d'une interaction RestFul
- Savoir interagir 2 micro-service avec *RestTemplate*

Description :
Dans cet atelier, 2 services RestFul Spring indépendants sont fournies :

- ***OrderService*** : Gére des commandes de produits
- ***TicketService*** : Gère les préparations de commande. Préparer les produits d'une commande pour une livraison

Au début de l'atelier, les 2 services sont indépendants. Le travail demandé est de compléter *OrderService* afin de créer un ticket lorsqu'une commande est créée

3.1 Découverte des APIs Restful

Importer les 2 projets Maven dans votre IDE
Visualiser le *pom.xml* puis les démarrer

Accéder à la documentation des APIs :

- ***TicketService*** : http://localhost:8080/swagger-ui.html
- ***OrderService*** : http://localhost:8081/swagger-ui.html

Voici les corps Json pour tester les requêtes POST
OrderService : Création de commande

```
POST {
  "discount": 5,
  "paymentInformation": {
    "fromAccount": "FROM",
    "toAccount": "TO",
    "amount": 1000
  },
  "deliveryInformation": {
    "pickAddress": {
      "rue": "string",
      "ville": "string",
      "codePostal": "string"
    },
    "deliveryAddress": {
      "rue": "string",
      "ville": "string",
      "codePostal": "string"
    }
  },
  "orderItems": [
    {
      "refProduct": "REF",
      "price": 100,
      "quantity": 10
    }
  ]
}
```

TicketService : Création de ticket

```
POST {
  "orderId": 1,
  "products": [
    {
      "reference": "AREF",
      "quantity": 2
    }
  ]
}
```

Visualiser les sources et exécuter les tests

3.2 Implémentation appel Rest dans order-service

Les classes du domaine sont différentes entre les services.
Une caractéristique et un inconvénient des APIs RestFul est que le client doit s'adapter à l'API qu'il veut consommer en créant des classes spécifiques.

Dans notre exemple, le projet *OrderService doit* créer les classes DTO nécessaires pour l'interaction avec TicketService
Par exemple :

- ***org.formation.TicketDto*** créé à partir d'une classe *Order*
- ***org.formation.TicketProductRequest*** créé à partir d'une classe *ProductRequest* :

```
@Data
public class TicketDto {
	@Min(1)
   long orderId;
   TicketProductRequest[] products;

   public TicketDto(Order order) {
	this.orderId = order.getId();
	List<TicketProductRequest> list =
order.getOrderItems().stream().map(TicketProductRequest::new).toList();
	products = new TicketProductRequest[list.size()];
	list.toArray(products);

  }
}
```

```
@Data
public class TicketProductRequest {

	@NotNull
	private String reference;
	@Min(1)
	private int quantity;

	public TicketProductRequest(OrderItem orderItem) {
		this.reference = orderItem.getRefProduct();
		this.quantity = orderItem.getQuantity();
	}

}
```

Editer la classe *org.formation.service.OrderService* pour implémenter l'appel *REST*

- Se faire injecter un ***RestTemplateBuilder*** afin de créer un ***RestTemplate***

- Compléter la méthode *createOrder()* pour effectuer une requête POST vers *TicketService*

La classe *OrderService* peut devenir comme suit :

```
@Service
@Transactional
public class OrderService {
	private final OrderRepository orderRepository;
	private final RestTemplate restTemplate;

	public OrderService(OrderRepository orderRepository, RestTemplateBuilder
restTemplateBuilder) {
		this.orderRepository = orderRepository;
		this.restTemplate = restTemplateBuilder.rootUri("http://localhost:8082").build();
	}

	public Order createOrder(Order order) {
		order.setDate(Instant.now());
		order.setStatus(OrderStatus.PENDING);

		order = orderRepository.save(order);

		restTemplate.postForEntity("/api/tickets", new TicketDto(order), Object.class);

		return order;
	}
}
```

Pour valider votre travail, vérifier que le test de *OrderControllerTest* passe lorsque *TicketService* est démarré

Atelier 4 : Interaction asynchrone via Message Broker

Objectifs de l'atelier :

- Comprendre les avantages des interactions asynchrone
- Avoir un premier aperçu du support Spring pour Kafka
- Implémenter le *Transactional Outbox Pattern*

Description :
Dans cet atelier, nous reprenons les 2 services RestFul Spring indépendants de l'atelier précédent:

Au début de l'atelier, les 2 services sont indépendants. Le travail demandé est de compléter *OrderService* afin de créer un ticket lorsqu'une commande est créée

4.1 Reprise et mise en place des projets

Reprendre les sources fournis du répertoire ***TPS/3_RestFul*** pour revenir dans l'état de départ.

Ajout les dépendances Kafka dans les 2 projets
Spring → Add Starter → Spring Apache Kafka

Visualiser le fichier ***TPS/4_Messaging/docker-compose-dev.yml***. Il permet de démarrer un cluster Kafka simple nœud ainsi qu'un outil d'administration Redpanda

Démarrer via :

```
docker-compose -f kafka-dev.yml up -d
```

Vérifier le bon démarrage en accédant à la console redpanda
http://localhost:9090

4.2 Implémentation Transactional Outbox Pattern

Dans le projet ***OrderService***, toute création ou modification de l'entité *Order* doit enregistrer l'événement dans une table.

Une tâche s'exécutant régulièrement scrute la table d'événement et pour chaque événement trouvé envoi un message vers le topic order de Kafka

Le pattern garantit qu'une transaction englobe la mise à jour de la table *Order* et la table *Event*. On suppose que les événements stockés dans la table Event finiront pas être envoyé à Kafka.

4.2.1 Enregistrement d'événement lors de la création de commande

Créer un package ***org.formation.event*** et y créer les classes suivantes

Types d 'événements :
Créer une *enum* listant les événements gérés par l'application :

```
public enum OrderEventType {
   ORDER_CREATED,
```

```
    ORDER_PAID,
    ORDER_CANCELLED ,
    ORDER_DELIVERED
}
```

Définir l'entité OrderEvent
Créer une classe entité *OrderEvent* comme suit :

```
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class OrderEvent {

        @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
        long id;
        @NotNull
        private long orderId;
        private EventType type;
        @Column(length = 10000)
        private String payload;
}
```

La classe contient un identifiant généré, une référence à la commande, un type d'évènement et une charge utile payload qui sera renseigné au format json.

Définir l'interface de persistance *OrderEventRepository*

```
public interface OrderEventRepository extends JpaRepository<Event, Long> {
}
```

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 créés.

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);
```

4.2.2 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

L'implémentation pourrait être :

```
@Service
@Log
```

```
public class EventService {
   private static String ORDER_CHANNEL="orders";
   private final OrderEventRepository orderEventRepository;
   private final KafkaTemplate<Long, Event> kafkaTemplate;

   public EventService(OrderEventRepository orderEventRepository, KafkaTemplate<Long,
OrderEvent> kafkaTemplate ) {
      this. orderEventRepository = orderEventRepository;
      this.kafkaTemplate = kafkaTemplate;
   }

   @Scheduled(fixedDelay = 10l, timeUnit = TimeUnit.SECONDS)
   public void relayEvents() {
      orderEventRepository.findAll()
            .forEach(event -> {
               log.info("Sending event"+event);
               kafkaTemplate.send(ORDER_CHANNEL, event.getOrderId(), event);
               orderEventRepository.delete(event);
            });
   }
}
```

4.2.3 Configuration Kafka

Nous effectuons une configuration minimale de Kafka :

- Définition du ***bootstrap-server***
- Définition des **sérialiseurs** utilisés pour la clé et la charge utile du messages

```
spring:
  kafka:
    bootstrap-servers:
    - localhost:9094
    producer:
      value-serializer: org.springframework.kafka.support.serializer.JsonSerializer
      key-serializer: org.apache.kafka.common.serialization.LongSerializer

app:
  order-channel: orders
```

4.2.4 Test

Vérifier le bon fonctionnement en effectuant une requête POST via swagger.

Ensuite vous pouvez utiliser le script JMeter fourni qui permet de générer 500 commandes.

Accéder à la console Redpanda pour visualiser les messages

4.3 Consommation des messages

Nous avons pu produire des messages sans que les consommateurs soient démarrés, l'avantage fourni par le *Messaging Pattern*

Nous allons maintenant consommer les messages dans *TicketService*.

Dans le projet TicketService, nous implémentons un listener de topic qui réagit aux messages postés

dans le topic ***orders***.

Seule la cas de la création de commande est implémentée (qui doit provoquer la création du Ticket)

4.3.1 Mise en place des classes DTO

Les sérialiseurs utilisés nécessitent de déclarer côté consommateur une classe ***org.formation.event.OrderEvent***. (Regarder via Redpanda les entêtes des messages Kafka). Cette contrainte pourrait être levée, nous verrons cela dans la suite de la formation.

Créer une classe ***org.formation.event.OrderEvent*** comme suit :

```
@Data
public class OrderEvent {
   private long orderId;
   private String type;
   private String payload;
}
```

Le champ ***payload*** est une String JSON que nous aimerions adapter aux classes existantes de *TicketService* en particulier *OrderDto* déjà utilisé dans l'API REST.

Différentes techniques sont possibles, ici nous utilisons les annotations Jackson.
Annoter *OrderDto* comme suit :

```
public class OrderDto {
        @Min(1)
        @JsonAlias({"id","orderId"})
         long orderId;

         @JsonAlias({"products","orderItems"})
         ProductRequest[] products;
}
```

Et *ProductRequest* comme suit :

```
public class ProductRequest {

        @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
        @JsonIgnore
        private Long id;
        @NotNull
        @JsonAlias({"reference","refProduct"})
        private String reference;
        @Min(1)
        private int quantity;
}
```

4.3.2 Développement du gestionnaire de messages

Créer une classe ***org.formation.service.EventHandler*** et une méthode *handleOrderEvent(OrderEvent)* annotée par *@KafkaListener* qui précise le topic d'écoute et l'identité du consommateur.

```
@Service
```

```java
@Log
public class EventHandler {

   @Autowired
   TicketService ticketService;

   @Autowired
   ObjectMapper mapper;


   private int nbEvent=0;

   @KafkaListener(topics="#{'${app.channel.order-channel}'}", id="ticket-service")
   public void handleOrderEvent(OrderEvent orderEvent) throws JsonMappingException,
JsonProcessingException {
        log.info("Consuming orderEvent " + (nbEvent++) + " consumed");
      switch (orderEvent.getType() ) {
         case "ORDER_CREATED":
                OrderDto orderDto = mapper.readValue(orderEvent.getPayload(),
OrderDto.class);
            ticketService.createTicket(orderDto);
            break;

      }
   }
}
```

4.3.3 Configuration Kafka

La configuration Kafka consiste à

- Fournir une adresse pour ***bootstrap-servers***
- Déclarer les désérialiseurs utilisés
- Indiqué que le consommateur désire traiter les messages depuis le début
- Le nom du topic Kafka utilisé

```yaml
spring:
  kafka:
    consumer:
      auto-offset-reset: earliest
      key-deserializer: org.apache.kafka.common.serialization.LongDeserializer
      value-deserializer:
org.springframework.kafka.support.serializer.JsonDeserializer
      properties:
        spring.json.trusted.packages: '*'
    bootstrap-servers:
      - localhost:9094
app:
  channel:
    order-channel: 'orders'
```

4.3.4 Tests

Le test consiste à visualiser la trace de *TicketService* et de s'assurer que tous les messages soient traités.

Il faut noter que *OrderService* peut être arrêté lorsque *TicketService* consomme les messages et inversement (*TicketService* peut être arrêté lorsque *OrderService* produit des messages).

Les cycles de vie des 2 services sont complètement indépendants à la différence des interactions

synchrones.

Pour refaire le test, connecter vous à Redpanda et supprimer le groupe de consommateur associé au topic ***orders***

Atelier 5 : Cluster Kafka

5.1 Installation cluster

5.1.1 Création du cluster ID et formattage des répertoires de logs

Télécharger et dézipper la distribution de Kafka dans un répertoire ***$KAFKA_DIST***

Créer un répertoire ***$KAFKA_LOGS*** qui stockera les messages des 3 brokers

Créer un répertoire ***$KAFKA_CLUSTER*** et 3 sous-répertoires : ***broker-1***, ***broker-2***, ***broker-3***

Copier le fichier de la distribution *$KAFKA_DIST/config/kraft/server.properties* dans les 3 sous-répertoires de $***KAFKA_CLUSTER***

Éditer les 3 fichier ***server.properties*** afin de modifier les propriétés :

- ***node.id***
- ***listeners***
- ***advertised.listeners***
- ***controller.quorum.voters***
- ***log.dir*** à ***$KAFKA_LOGS/broker-<id>***

Générer un id de cluster :

$KAFKA_DIST/bin/kafka-storage.sh random-uuid

Utiliser l'ID du cluster pour formatter les 3 répertoires

Pour chaque broker, formatter son répertoire de log avec :

bin/kafka-storage.sh format -t <cluster_id> -c $KAFKA_CLUSTER/broker-<id>/server.properties

5.1.2 Mise au point d'un script de démarrage/arrêt

Mettre au point un script sh permettant de démarrer les 3 brokers en mode daemon :

Exemple :

```
#!/bin/sh


export JAVA_HOME=/usr/lib/jvm/java-17-openjdk-amd64/
export KAFKA_DIST=/home/dthibau/Formations/SpringKafka/MyWork/kafka-dist/kafka_2.13-3.2.1
export KAFKA_CLUSTER=/home/dthibau/Formations/SpringKafka/github/solutions/kafka-cluster
export KAFKA_LOGS=/home/dthibau/Formations/SpringKafka/MyWork/kafka-logs


$KAFKA_DIST/bin/kafka-server-start.sh -daemon $KAFKA_CLUSTER/broker-1/server.properties
$KAFKA_DIST/bin/kafka-server-start.sh -daemon $KAFKA_CLUSTER/broker-2/server.properties
$KAFKA_DIST/bin/kafka-server-start.sh -daemon $KAFKA_CLUSTER/broker-3/server.properties
```

Visualiser les traces de démarrages :

tail -f $KAFKA_DIST/logs/server.log

Mettre au point un script d'arrêt

```
#!/bin/sh


export KAFKA_DIST=/home/dthibau/Formations/SpringKafka/MyWork/kafka-dist/kafka_2.13-3.2.1
$KAFKA_DIST/bin/kafka-server-stop.sh
```

Effectuer les vérifications de création de topic et envoi/réception de messages

```
$KAFKA_DIST/bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic test

$KAFKA_DIST/bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic test

$KAFKA_DIST/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning
```

5.2 Utilitaires

6.1.2 Production de messages

Créer un topic ***testing*** avec 5 partitions et 2 répliques :

```
$KAFKA_DIST/bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 2 --partitions 5 --topic testing
```

Lister les topics du cluster avec

```
$KAFKA_DIST/bin/kafka-topics.sh --list --bootstrap-server localhost:9092
```

Visualiser les propriétés du topic *testing*

```
$KAFKA_DIST/bin/kafka-topics.sh --describe --topic testing --bootstrap-server localhost:9092
```

Visualiser les répertoires de logs sur les brokers :

```
$KAFKA_DIST/bin/kafka-log-dirs.sh  --bootstrap-server localhost:9092 --describe
```

Démarrer un producteur de message

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

Saisir quelques messages de la forme <***key***> ***:***<***value***>. Par exemple

```
>1:hello

>2:world
```

6.1.3 Consommation des messages a posteriori

Dans une autre fenêtre, Consommer les messages de *testing* depuis le début

```
$KAFKA_DIST/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic testing --property "parse.key=true" --property "key.separator=:" --from-beginning
```

6.1.4 Visualisation des offsets

Dans une 3ème fenêtre, Lister les groupes de consommateurs

```
$KAFKA_DIST/bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list
```

Accéder au détail du groupe de consommateur en lecture sur le topic ***testing***

```
$KAFKA_DIST/bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group <group-id>
```

5.3 Outils graphiques

Installer l'outil graphique de votre choix

Exemple akhq

Télécharger une distribution d'akhq (akhq-all.jar)

Récupérer le fichier de configuration fourni ***application-basic.yml***

Exécuter le serveur via :
java -Dmicronaut.config.files=./application-basic.yml -jar akhq-0.21.0-all.jar

Exemple Kafka Magic :

```
docker run -d --rm --network=host digitsy/kafka-magic
```

Exemple Redpanda Console :

```
docker run --network=host \
  -e KAFKA_BROKERS=localhost:9092 \
  docker.redpanda.com/redpandadata/console:latest
```

Ateliers 6 : Apache Kafka et ses APIS

6.1 Producer API

6.1.1 Démarrage du cluster via docker-compose

Optionnel si installation cluster précédente réussie

Visualiser le fichier ***docker-compose.yml*** qui définit 3 brokers Kafka et le service Redpanda

Démarrer la stack avec

```
docker-compose up -d
```

Observer les logs de démarrages

```
docker-compose logs
```

Accéder à Redpanda et vérifier les statuts des brokers

```
http://localhost:9090/overview
```

6.1.2 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

6.1.3 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

6.1.4 Construction d'un exécutable et comparaison des modes d'envoi

Supprimer le topic et le recréer via Redpanda avec un nombre de *partitions=3* et un *replication-factor=2*

Construire un jar exécutable avec :
mvn package

Faites 3 démarrage en ligne de commande et comparer les temps d'exécution

```
java -jar target/producer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 10 10000 10 0
java -jar target/producer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 10 10000 10 1
java -jar target/producer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 10 10000 10 2
```

6.2 : Consumer API

Reprendre le projet Maven ***$TP_DATA/6_KafkaAPIs/6.2_Consumer/KafkaConsumer***

6.2.1 Implémentation

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

6.2.2 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

6.2.3 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);
}
```

6.2.4 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

6.3 Schema Registry et Avro

6.3.1 Démarrage de Confuent Registry

Visualiser le fichier ***TPS/6_KafkaAPIS/6.3_Schema/docker-compose.yml*** fourni

Démarrer la stack

Accéder à http://*localhost:9091/subjects*

Vérifier que Redpanda se connecte correctement à *Schema Registry*

Consulter le mode de compatibilité du Schema Registry

6.3.2 Producteur de message

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

Dépendances
Reprendre ***TPS/6_KafkaAPIs/6.3_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:9091/subjects

Puis à http://localhost:9091/schemas/

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

6.3.3 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

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

6.3.4 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

6.3.5 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*

6.4 : 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.

6.4.1 Reprise du projet

Reprendre le projet de ***TPS/6_KafkaAPIs/6.4_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

6.4.2 Processeur et sink

Ajouter au KStream ***coursierStream*** construit à partir du topic *position* :
- Un processeur filtrant les événements supérieurs à la latitude 48.8
- Un sink qui déverse les événements vers le topic ***position-nord***

Avec l'exemple du cours, écrire la classe principale qui effectue le traitement voulu

Ateliers 7 Configuration cluster et topic

7.1 : Garanties de livraison At Least Once

Objectifs :

Mettre en évidence la garantie *At Least Once.*

Description

On utilisera un cluster de 3 nœuds avec un topic de 3 partitions, un mode de réplication de 2 et un *min.insync.replica* de 1.

Le scénarios de test envisagé :

- Ré-équilibrage des consommateurs

Les métriques surveillés

- Côté consommateur : Doublon ou messages loupés

7.1.1 Configuration topic position :

Supprimer le topic ***position***

Le recréer avec

```
$KAFKA_DIST/bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 3 --partitions 3 --topic position
```

Si vous utilisez docker obtenir auparavant un shell sur un container

```
docker exec -it <container-id> /bin/bash
```

La distribution kafka est dans le répertoire ***/opt/bitnami/kafka/***

Vous pouvez également utiliser la console d'administration

Positionner le *min.insync.replicas à 2 (via la console d'administration)*

Vérifier l'affectation des partitions et des répliques via :

```
$KAFKA_DIST/bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic position
```

7.1.2 Tests

Récupérer les programmes fournis ***SpringConsumer*** et *SpringProducer*

Vérifier la configuration de *SpringConsumer* dans *src/main/resources/application.yml* et l'adapter à votre environnement.

Attention : ***logging.level*** doit être à debug lors des tests

Construire l'application via

```
./mvnw package
```

Dans 2 terminal distinct, démarrer 2 fois l'application :

```
java -jar target/SpringConsumer-0.0.1-SNAPSHOT.jar --
spring.kafka.listener.concurrency=1 >> log1.csv
```

```
java -jar target/SpringConsumer-0.0.1-SNAPSHOT.jar --
spring.kafka.listener.concurrency=1 >> log2.csv
```

Vérifier la configuration de *SpringProducer* dans *src/main/resources/application.yml* et l'adapter à votre environnement.

Construire l'application via

```
./mvnw package
```

Dans un autre terminal, démarrer 1 producteur multi-threadé :

```
java -jar SpringProducer.jar --spring.kafka.producer.enable.idempotence=true
```

Appuyer sur Entrée pour démarrer la production de 10000 messages

Pendant la consommation des messages, arrêter et redémarrer un consommateur (plusieurs fois)

7.1.3 Visualisation résultat :

Concaténer les fichiers résultas :

```
cat log1.csv >> cat log2.csv >> log.csv
```

Un utilitaire ***check-logs*** est fourni permettant de détecter les doublons ou les offsets perdus.

```
java -jar check-logs.jar <log.csv>
```

7.2 Exactly Once

Objectifs : Implémenter une sémantique Exactly Once avec les transactions Kafka

Description :

7.2.1 Envoi d'un lot de message dans une transaction

Reprendre le projet KafkaProducer et modifier la configuration du *producer* dans ***KafkaProducerThread*** en ajoutant les 2 propriétés suivantes :

```
kafkaProps.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "tx-position");
kafkaProps.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true");
```

Modifier le code de méthode run() afin :

- D'initialiser le mode transactionnel :
  producer.initTransaction()
- De délimiter une nouvel transaction tous les 10 messages :
  producer.beginTransaction()
  …

producer.commitTransaction()

Une fois votre programme mis au point construire le projet et exécuter le programme avec 1 thread et 105 messages.

```
mvn package
```

```
java -jar target/producer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 1 105 10 1
```

Observer les offsets

7.2.3 Consommateur read_committed

Reprendre le projet *KafkaConsumer*.

Modifier la configuration du consumer dans ***KafkaConsumerThread*** en ajoutant la propriétés suivante :

```
kafkaProps.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_committed");
```

Lancer une production de 105 messages Consommer les messages et visualiser le lag dans la console RedPanda

7.2.4 Traitement et production de messages

Nous voulons rajouter une information de distance par rapport à un point d'origine dans la classe Coursier.

Modification des classes du domaine

Ajouter la propriété `private Double distance;` et les getters/setters dans la classe Coursier

Ajouter les méthodes permettant de calculer la distance entre 2 Position dans la classe Position :

```
public Double distance(Position origin) {

	long R = 6371; // Radius de la terre en km
	var dLat = deg2rad(origin.getLatitude() - this.latitude);
	var dLon = deg2rad(origin.getLongitude() - this.longitude);
	var a = Math.sin(dLat / 2) * Math.sin(dLat / 2)
			+ Math.cos(deg2rad(this.latitude)) *
  Math.cos(deg2rad(origin.latitude)) * Math.sin(dLon / 2) * Math.sin(dLon / 2);
	var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
	return R * c; // Distance en km

}

// Degré vers Radius
private double deg2rad(double degre) {
	return degre * (Math.PI / 180);
}
```

Ajouter également la classe ***JsonSerializer*** du projet KafkaProducer car le consommateur va également produire des messages

Boucle de poll

Lors de la boucle de poll nous construisons une liste de Coursier avec la distance par rapport à une point d'origine calculée

```
List<Coursier> coursiers = new ArrayList<>();
for (var record : records) {
      nbMessages++;
      Coursier c = record.value();
      c.setDistance(c.getCurrentPosition().distance(origin));
      coursiers.add(c);
      Thread.sleep(10);
}
```

Modifier la configuration du consumer dans ***KafkaConsumerThread*** en ajoutant la propriétés suivante :

```
kafkaProps.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,"false");
```

Production de messages et envoi des offsets à committer

Initialiser un producer *KafkaProducer<String, Coursier>* transactionnel

```
private void _initProducer() {
      Properties kafkaProps = new Properties();
      kafkaProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:19092,localhost:19093,localhost:19094");
      kafkaProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                  "org.apache.kafka.common.serialization.StringSerializer");
      kafkaProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                  "org.formation.JsonSerializer");
      kafkaProps.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "tx-distance");
      kafkaProps.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true");

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

}
```

Dans le code de la méthode run() , initialiser les transactions

Puis pour chaque lot de messages reçus envoyer dans une transaction :

- les coursiers avec l'information de distance vers un topic de sortie.
- Les offsets à committer à Kafka

```
producer.beginTransaction();

coursiers.forEach((c) -> producer.send(new ProducerRecord<String,
Coursier>(OUTPUT_TOPIC, c.getId(), c)));

Map<TopicPartition, OffsetAndMetadata> offsetsToCommit = new HashMap<>();

for (TopicPartition partition : records.partitions()) {
     List<ConsumerRecord<String, Coursier>> partitionedRecords =
records.records(partition);
     long offset = partitionedRecords.get(partitionedRecords.size() -
1).offset();

     offsetsToCommit.put(partition, new OffsetAndMetadata(offset + 1));
}

System.out.println("Sending offsets to transaction " + offsetsToCommit);
producer.sendOffsetsToTransaction(offsetsToCommit, consumer.groupMetadata());

producer.commitTransaction();
```

Dans les exceptions gérés, effectuer un `producer.abortTransaction();`

7.2.5 Tests

Démarrer le consommateur, lancer une production d'une centaine de messages, observer la console du consommateur et le nombre de messages dans le topic de sortie.

Visualisez également le topic interne lié aux transition

7.3 Configuration de la rétention

Visualiser les segments et apprécier la taille.

Si option docker :

```
docker exec -it <container-id> /bin/bash
```

```
ls -al bitnami/kafka/data
```

Pour le topic ***position,*** modifier le ***segment.bytes*** à 512ko et diminuer le ***retention.bytes*** afin de voir des segments disparaître

Utiliser ***KafkaProducer*** pour générer des messages

Ateliers 8. Spring Kafka

8.1 KafkaTemplates

Objectifs :

- Utiliser quelques déclinaisons de KafkaTemplate proposées par Spring-Kafka

8.1.1 KafkaRoutingTemplate

Récupération du projet et description

Récupérer le projet ***TPS/8_SpringKafka/8.1_Templates/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

Configuration Beans Kafka

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(2)).build();
}
```

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());
   }
});;
```

Tests
En utilisant l'interface Swagger, voir la tolérance aux pannes en arrêtant des brokers puis provoquer une erreur *NOT_ENOUGH_REPLICA*

Pour créer une livraison :

```
POST {
  "pickAddress": {
    "rue": "string",
    "ville": "string",
    "codePostal": "string"
  },
  "deliveryAddress": {
    "rue": "string",
    "ville": "string",
    "codePostal": "string"
  }
}
```

Pour indiquer un position à un coursier :

```
PATCH /api/coursier/1/move
```

8.1.2 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***

Reprise du projet PaymentService

Reprendre le projet ***TPS/8_SpringKafka/8.1_Templates/PaymentService***

Vérifier son démarrage et son API Rest :
http://localhost:8084/swagger-ui/index.html

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

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;
```

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);
```

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 créer une commande via l'interface Swagger de ***OrderService*** :

POST http://localhost:8081/api/orders

```
{

  "discount": 0,
  "paymentInformation": {
    "fromAccount": "string",
    "toAccount": "string",
    "amount": 10
  },
  "deliveryInformation": {
    "pickAddress": {
      "rue": "string",
      "ville": "string",
      "codePostal": "string"
    },
    "deliveryAddress": {
      "rue": "string",
      "ville": "string",
      "codePostal": "string"
    }
  },
  "orderItems": [
    {
      "refProduct": "string",
      "price": 10,
      "quantity": 1,
      "total": 10
    }
  ],
  "total": 10
}
```

8.2 Consommation de message et @KafkaListener

Objectifs :

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

8.2.1 Commit manuel à chaque enregistrement

Modifier la configuration du projet ***TicketService*** pour spécifier un Commit manuel à chaque enregistrement reçu

```
spring:
  kafka:
    listener:
      ack-mode: manual-immediate
```

Dans la classe ***org.formation.service.Eventhandler***, modifier la signature de la méthode annotée par *@KafkaListener* afin d'ajouter un argument de type ***Acknowledgement***

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;

   }
}
```

8.2.2 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 recçoit 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

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

```
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
```

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);
        });
}
```

Tests

Récupérer le projet ***TPS/8_SpringKafka/8.2_Listener/PositionService*** qui offre une interface REST pour publier des messages

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

8.3 Transaction et Exactly Once

Objectifs :

Envoyer 2 messages en une seule transaction

Appliquer la sémantique Exactly Once à l'interaction *Order* → *PaymentService* → *Order*

8.3.1 Producteur transactionnel

Dans ***org.formation.service.EventService*** lors d'un événement *OrderEventType.ORDER_CREATED*, faire également une requête de paiement.

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 ***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")***

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 :

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

Effectuer une requête POST et visualiser les traces

8.3.2 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 :

```
spring:
  kafka:
    producer:
```

```
      transactionIdPrefix: "payment-tx-"
    consumer:
      isolation-level: READ_COMMITTED
```

Ajouter également

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

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

8.3.3 Test via JMeter

Utiliser le fichier JMeter ***TPS/8_SpringKafka/8.3_Transactions/CreateOrderTransacation.jmx*** pour tester des transactions validées et non validées

8.4 Sérialisation

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

8.4.1 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, 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

8.4.2 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");
	}
}
```

8.4.3 Test

Tester les 2 endpoints de *PositionService*

8.5 Traitement des Exceptions

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

8.5.1 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/8_SpringKafka/8.5_Exceptions/Commandes.jmx*** pour générer un certain nombre de messages de type commande

Configuration par défaut

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

Configuration d'un gestionnaire par défaut

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

```
@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 if ( offset%10 == 0 && retry < 3 ) {
                retry++;
                throw new RuntimeException("Boom");
        } else if ( offset%10 == 0 && retry >= 3 ) {
                retry=0;
        }else {
                process++;
        }

}
```

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*

8.5.2 Erreur de sérialisation :

Comportement par défaut

Récupérer le service ***TPS/8_SpringKafka/8.5_Exceptions/PaymentBadService*** et visualiser son code et les sérialiseurs utilisés

Démarrer également ***PaymentService***

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

POST http://localhost:8086/api/bad

Visualiser la console de *PaymentService*

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

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 9: Sécurité

9.1 Séparation des échanges réseaux

Installation à partir de l'archive

Modifier les fichiers ***server.properties*** des 3 brokers afin de créer 3 listeners *PLAIN_TEXT* dénommé PLAIN_TEXT, CONTROLLER et EXTERNAL en leur affectant des ports différents

Pour l'instant utiliser des communications en clair pour les 3

3 propriétés de configuration doivent être mises à jour :

- listeners
- advertised.listeners
- listener.security.protocol.map

Avec un programme précédent utiliser le listener EXTERNAL

Installation docker-compose

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 Mise en place de SSL pour crypter les données

9.2.1 Génération keystore et truststore

Créer nouveau répertoire ***ssl*** permettant de stocker les keystore

Y déposer le script ***TPS/9_Security/9.2_TLS/generate-ssl.sh*** fourni (Provient de Bitnami)

Faites attention lors des réponses aux questions de l'assistant :

- Lorsque vous êtes invité à entrer un mot de passe, utilisez le même pour tous.
  Par exemple : *secret*
- Définissez les valeurs Common Name ou FQDN sur le nom d'hôte de votre conteneur Apache Kafka, par ex. kafka.exemple.com. Après avoir saisi cette valeur, lorsque vous êtes invité "Quel est votre nom et prénom ?", saisissez également cette valeur.

A la fin de l'opération, vous devez avoir 2 fichiers *.jks* :

- ***keystore/kafka.keystore.jks*** : Certificats utilisé par les brokers
- ***truststore/kafka.truststore.jks*** : Truststore pour les serveurs

Vous pouvez vérifier les contenus des store avec :

```
keytool -v -list -keystore keystore/kafka.keystore.jks
```

9.2.2 Configuration pour SSL appliqué aux communications inter-broker

Option Archive

Configurer les fichiers *server.properties afin d'a*jouter un nouveau SSL, mappé sur le protocole SSL et l'utiliser pour la communication inter-broker.

Les propriétés à modifier sont :

- `listeners`
- `inter.broker.listener.name`
- `advertised.listeners`
- `listener.security.protocol.map`

Configurer le listener SSL et les propriétés SSL suivante dans les fichiers *server.properties*

```
ssl.keystore.location=<your-env>/ssl/server.keystore.jks
ssl.keystore.password=secret
ssl.key.password=secret
ssl.truststore.location=<your-env>/ssl/server.truststore.jks
ssl.truststore.password=secret
security.inter.broker.protocol=SSL
ssl.endpoint.identification.algorithm=
ssl.client.auth=none
```

Démarrer le cluster kafka et vérifier son bon démarrage

Dans les traces doivent apparaître :

```
[2023-08-14 10:10:13,308] INFO [SocketServer listenerType=BROKER,
nodeId=2] Started data-plane acceptor and processor(s) for
endpoint : ListenerName(SSL) (kafka.network.SocketServer)
```

Option Docker

Dans l'environnement docker les noms DNS ne sont pas *localhost*. Les certificats générés précédemment ne sont valable.

Donc rester en plaintext pour la communication inter-broker

Solution possible :

A partir du docker-compose, il faut créer 3 certificats différents pour ***kafka-0, kafka-1, kafka-2***

Et les importer dans le truststore

Déposer les keystore et trustore dans `/opt/bitnami/kafka/config/certs/` en faisant un montage de répertoire

Puis effectuer une configuration par variable d'environnement comme suit :

```
environment:
- KAFKA_CFG_NODE_ID=0
- KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=0@kafka-0:9093,1@kafka-1:9093,2@kafka-2:9093
- KAFKA_KRAFT_CLUSTER_ID=abcdefghijklmnopqrstuv
- KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,CONTROLLER://:9093,EXTERNAL://:19092,SSL://:9095
- KAFKA_CFG_INTER_BROKER_LISTENER_NAME=SSL
- KAFKA_CFG_CONTROLLER_LISTENER_NAMES=CONTROLLER
- KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka-0:9092,EXTERNAL://localhost:19092,SSL://kafka-0:9095
-
KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=CONTROLLER:PLAINTEXT,EXTERNAL:PLAINTEXT,PLAINTEXT:PLAINTEXT,SSL:SSL
- KAFKA_CFG_SSL_KEYSTORE_LOCATION=/opt/bitnami/kafka/config/certs/kafka.keystore.jks
- KAFKA_CFG_SSL_KEYSTORE_PASSWORD=secret
- KAFKA_CFG_SSL_KEY_PASSWORD=secret
- KAFKA_CFG_SSL_TRUSTSTORE_LOCATION=/opt/bitnami/kafka/config/certs/kafka.truststore.jks
- KAFKA_CFG_SSL_TRUSTSTORE_PASSWORD=secret
- KAFKA_CFG_SSL_CLIENT_AUTH=none
- KAFKA_CFG_SSL_ENDPOINT_IDENTIFICATION_ALGORITHM=
```

9.2.3 Configuration pour SSL appliqué aux communications externes

Utiliser également SSL pour le listener EXTERNAL, il suffit de modifier la propriété :

- listener.security.protocol.map

9.2.4 Accès client via SSL

Mettre au point un fichier ***client-ssl.properties*** avec :

security.protocol=SSL

ssl.truststore.location=/home/dthibau/Formations/SpringKafka/github/solutions/ssl/truststore/kafka.truststore.jks
ssl.truststore.password=secret

Vérifier la connexion cliente avec par exemple

```
$KAFKA_DIST/bin/kafka-console-producer.sh --broker-list localhost:EXTERNAL_PORT
--topic ssl --producer.config client-ssl.properties
```

Configurer le projet Spring ***PositionService*** pour configurer le client avec ssl.

Dans KafkaConfig, rajouter les propriétés de configuration suivantes :

```
props.put("security.protocol", "SSL");
props.put("ssl.truststore.location", "<your-config>/kafka.truststore.jks");
props.put("ssl.truststore.password", "secret");
```

Tester un envoi de message, vérifier la bonne configuration du *KafkaProducer* et 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";
};
```

Modifier le script de démarrage afin qu'il utilise le fichier :

```
export KAFKA_OPTS="-Djavax.net.debug=ssl:handshake:verbose
-Djava.security.auth.login.config=<your-env>/kafka_server_jaas.conf"
```

Modifier les fichiers *server.properties* afin que la communication inter-broker utilise le listener ***SASL_SSL***.

Vous devez modifier les propriétés de configuration :

- *listeners*
- *inter.broker.listener.name*
- *advertised.listeners*

Ajouter également les configurations :
sasl.mechanism.inter.broker.protocol=PLAIN
sasl.enabled.mechanisms=PLAIN

Redémarrer le cluster et vérifier son bon démarrage

9.3.2 Authentification client

Configurer le listener EXTERNAL pour qu'il utilise également SASL_SSL comme protocole

Mettre à jour le fichier ***client-ssl.properties*** comme suit :
security.protocol=SASL_SSL
sasl.mechanism=PLAIN
*sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required *
*username="alice" *
password="alice-secret";

Tester avec :
$KAFKA_DIST/bin/kafka-topics.sh --bootstrap-server localhost:19092 --list --command-config ssl/client-ssl.properties

Configurer le projet Spring ***PositionService*** pour configurer le client avec ssl.
Dans KafkaConfig, rajouter les propriétés de configuration suivantes :

```
props.put("security.protocol", "SASL_SSL");
props.put("ssl.truststore.location", "<your-config>/kafka.truststore.jks");
props.put("ssl.truststore.password", "secret");
props.put("sasl.jaas.config","org.apache.kafka.common.security.plain.PlainLoginModule
required username=\"alice\" password=\"alice-secret\";");
props.put("sasl.mechanism", "PLAIN");
```

Tester un envoi de message, vérifier la bonne configuration du *KafkaProducer* et la production de message

9.4 ACL

9.4.1 Configuration brokers

Activer les ACL avec la classe ***org.apache.kafka.metadata.authorizer.StandardAuthorizer***

Définir les super users et la règle allow.everyone.if.no.acl.found=true

Indiquer que le contrôleur doit s'identifier

listener.name.controller.ssl.client.auth=required

authorizer.class.name=org.apache.kafka.metadata.authorizer.StandardAuthorizer
super.users=User:ANONYMOUS,User:admin,User:alice
allow.everyone.if.no.acl.found=true

Démarrer le cluster et vérifier son bon démarrage

Tester l’envoi d’un message par ***PositionService***

A Voir également les liens suivants :

https://github.com/bitnami/containers/issues/23237

https://stackoverflow.com/questions/74671787/kafka-cluster-using-kraft-mtls-and-standardauthorizer-not-starting-up-getting

9.4.2 Définition ACLs

Définir une ACL via Redpanda Console interdisant à l'utilisateur alice d'écrire sur les topics.

Tester le refus d'envoi de message par ***PositionService***