Architecture Kafka Stream

TP : Développement d'une application avec Java, Spring et Kafka Stream

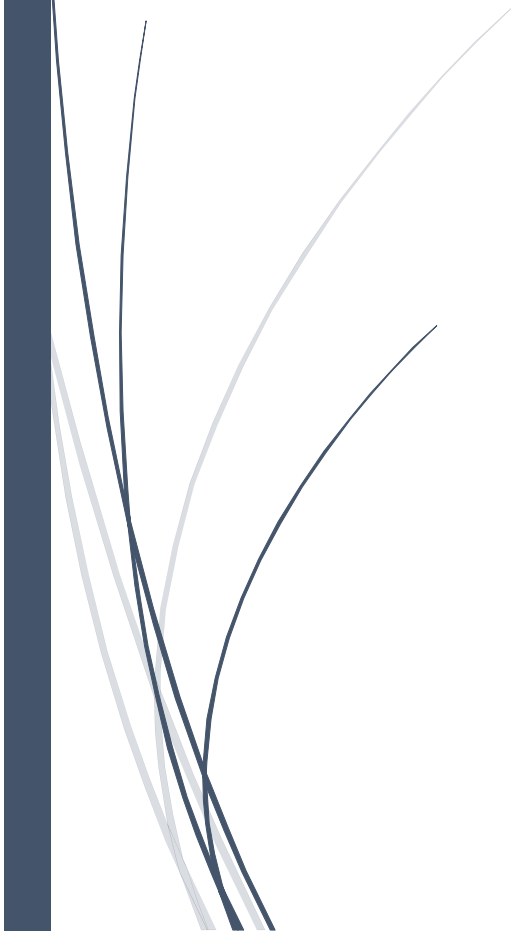


Table des matières

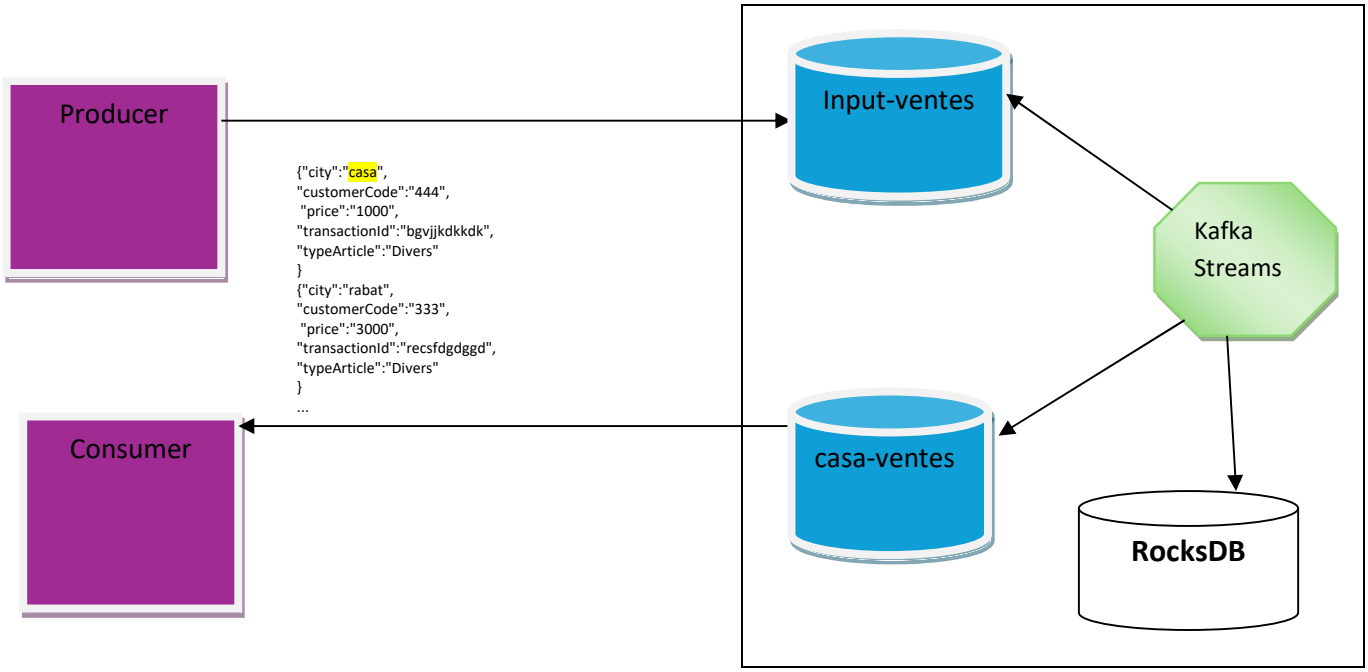
I. Objectif du TP

- Expliquer l'architecture de ***Kafka Streams***.
- Publier et consommer un objet complexe avec Spring et Kafka.
- Manipuler un flux de données moyennant l'api Kafka Streams.
- Développer une application avec Java, Spring et Kafka Stream.

II. Objet du TP

Nous allons voir à travers cet exemple comment envoyer des objets de type VenteDto (composé des éléments suivants : city, customerCode, price, transactionId et typeArticle) au Token « ***input-ventes*** », filtrer uniquement les ventes réalisées au niveau de la ville de casablanca, transmettre les flux filtrés au Topic « ***casa-ventes »*** et enregistrer pour chaque client le montant total des ventes réalisés.

Le schéma suivant illustre les traitements effectués par Kafra Streams :



III. Prérequis

- IntelliJ IDEA ;
- JDK version 17 ;
- Une connexion Internet pour permettre à Maven de télécharger les librairies.

NB : *Ce TP a été réalisé avec IntelliJ IDEA 2023.2.3 (Ultimate Edition).*

IV. Kafka Streams

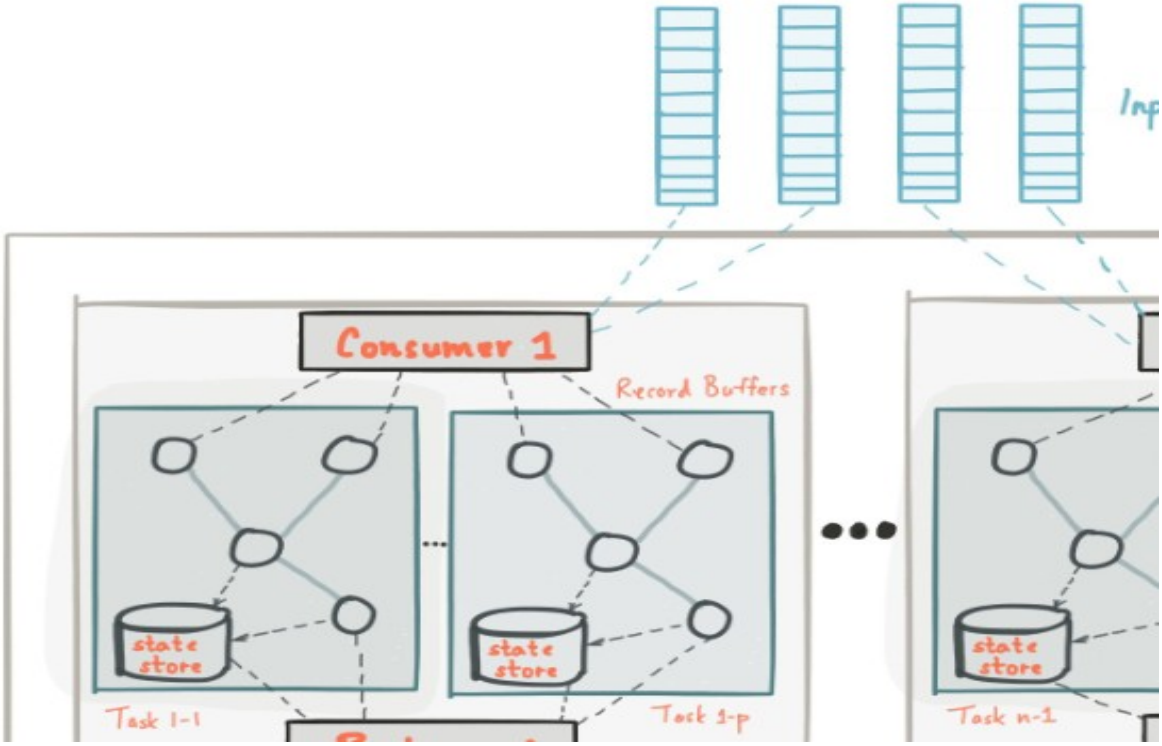
a. Kafka Sreams, c'est quoi ?

Kafka Streams est une bibliothèque Java développée par Apache Kafka pour créer des applications de traitement de flux (stream processing) de manière distribuée, scalable et résiliente. Contrairement à un simple producteur ou consommateur Kafka, ***Kafka Streams*** permet d'effectuer des opérations complexes comme des transformations, des agrégations et des jointures sur des flux de données en temps réel.

b. Principales fonctionnalités de Kafka Streams

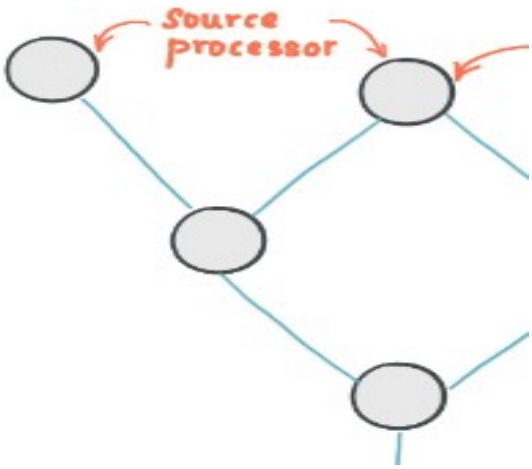
- ***Traitement en temps réel*** : Kafka Streams est conçu pour transformer, enrichir et analyser des données en temps réel au fur et à mesure qu'elles transitent par Kafka.
- ***API haut niveau*** : Kafka Streams fournit une API intuitive pour effectuer des transformations sur les données (exemple : filter, map, flatMap, groupBy, etc.).
- ***Sans serveur supplémentaire*** : Pas besoin de cluster distinct (comme *Apache Flink* ou *Spark Streaming*). L'application Kafka Streams s'exécute comme une application autonome.
- ***Etat local et fault tolerance*** : Kafka Streams peut gérer des états locaux (comme les agrégations ou les fenêtres temporelles) en stockant les données dans des State Stores (souvent basés sur RocksDB). Ces données peuvent être récupérées en cas de panne.
- ***Scalabilité et tolérance aux pannes*** : Kafka Streams utilise les partitions Kafka pour le parallélisme et peut se redimensionner dynamiquement en fonction du nombre d'instances d'une application.
- ***Interopérabilité avec Kafka*** : Kafka Streams s'intègre parfaitement avec Kafka en tant que source et destination. Vous pouvez lire les messages d'un topic, les transformer, puis les écrire dans un autre topic.
- ***Support pour les fenêtres temporelles*** : Kafka Streams facilite les traitements basés sur des fenêtres temporelles (exemple : "compter les événements par minute").

c. L'architecture de l'api Kafka Streams



Les composants principaux de l'api **Kafka Streams** sont :

- **Kafka Topics :**
  - ✓ Les topics sont les unités de stockage et de communication dans Kafka.
  - ✓ Kafka Streams consomme les données d'un ou plusieurs topics sources et peut produire des résultats dans d'autres topics.
- **Stream** : Un stream est un flux de données immuable et ordonné de messages tirés d'un topic Kafka. Chaque message est ***une paire clé-valeur***.
- **Stream Processing Topology** :



  - ✓ Une topology est une représentation logique d'une application Kafka Streams.

- ✓ Elle est composée de processeurs de flux (Stream Processors) connectés entre eux :
  - **Source Processor** : Point d'entrée des données, lisant les messages depuis les topics Kafka.
  - **Stream Processor** : Applique des transformations (mappage, filtrage, regroupement, etc.) sur les données.
  - **Sink Processor** : Envoie les résultats dans un ou plusieurs topics Kafka.

- **State Store** :
  - ✓ Kafka Streams utilise des ***State Stores*** pour conserver l'état intermédiaire nécessaire à certaines opérations (comme les jointures, les regroupements, ou les agrégations).
  - ✓ Ces magasins d'état sont intégrés à l'application et peuvent être sauvegardés dans Kafka pour la tolérance aux pannes.
- **Partitionnement et parallélisme** :
  - ✓ Kafka Streams exploite le partitionnement des topics pour permettre le traitement parallèle.
  - ✓ Chaque instance de l'application Kafka Streams traite une ou plusieurs partitions.
- **RocksDB** : Par défaut, Kafka Streams utilise ***RocksDB*** comme moteur de stockage local pour gérer les State Stores.

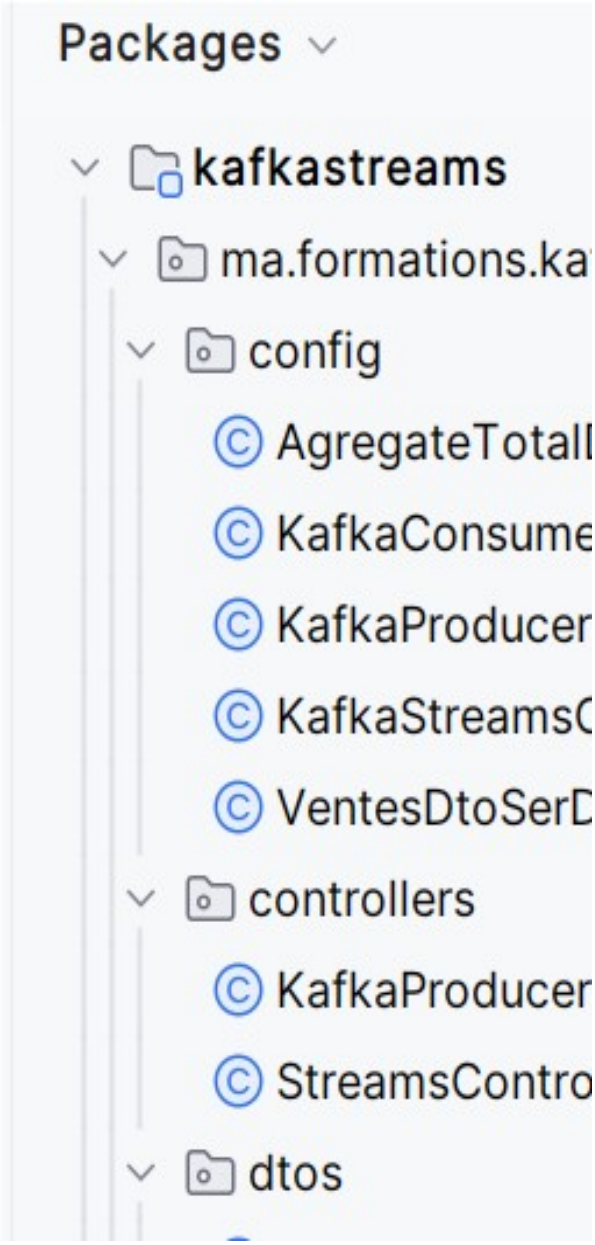
d. Cas d'utilisation

- Monitoring en temps réel : Analyser les logs d'une application pour détecter des anomalies.
- Traitement d'événements : Calculer des agrégats en temps réel, comme le nombre total de transactions par minute.
- Systèmes de recommandation : Générer des recommandations personnalisées en temps réel en fonction du comportement de l'utilisateur.
- ETL en temps réel : Extraire des données de Kafka, les transformer et les charger dans un autre système.
- Alertes : Déclencher des notifications en temps réel en fonction d'événements spécifiques (exemple : détection de fraudes).

V. Développer une application avec Java et Kafka

a. L'arborescence du projet

- Créer un projet Spring Boot (nommer le par exemple : kafkastreams) :



b. Le fichier pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.1.5</version>
    <relativePath/>
  </parent>
  <groupId>ma.formations.kafka.streams</groupId>
  <artifactId>kafkastreams</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>kafkastreams</name>
  <description>kafkastreams</description>
  <properties>
    <java.version>17</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-streams</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.kafka</groupId>
      <artifactId>spring-kafka</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.kafka</groupId>
      <artifactId>spring-kafka-test</artifactId>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
```

```
        <scope>provided</scope>
      </dependency>
    </dependencies>

    <build>
      <plugins>
        <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
          <configuration>
            <excludes>
              <exclude>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
              </exclude>
            </excludes>
          </configuration>
        </plugin>
      </plugins>
    </build>

</project>
```

c. Le fichier application.properties

```
server.port=8081
spring.kafka.bootstrap-server=localhost:9092
spring.kafka.topic.input=input-ventes
spring.kafka.topic.output=casa-ventes
spring.jackson.serialization.FAIL_ON_EMPTY_BEANS=false
```

d. La classe VentesDto

```
package ma.formations.kafka.streams.dtos;

import lombok.*;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class VentesDto {
   private String city;
   private String customerCode;
   private String price;
   private String transactionId;
   private String typeArticle;
}
```

e. La classe AggregateTotalDto

```
package ma.formations.kafka.streams.dtos;

import lombok.*;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class AggregateTotalDto {
  private long count;
  private double amount;
}
```

f. La classe VentesDtoSerDes

```
package ma.formations.kafka.streams.config;

import ma.formations.kafka.streams.dtos.VentesDto;
import org.apache.kafka.common.serialization.Serdes;
import org.springframework.kafka.support.serializer.JsonDeserializer;
import org.springframework.kafka.support.serializer.JsonSerializer;

public class VentesDtoSerDes extends Serdes.WrapperSerde<VentesDto>{
   public VentesDtoSerDes() {
      super(new JsonSerializer<>(), new JsonDeserializer<>(VentesDto.class));
   }
}
```

g. La classe AgregateTotalDtoSerDes

```
package ma.formations.kafka.streams.config;

import ma.formations.kafka.streams.dtos.AggregateTotalDto;
import org.apache.kafka.common.serialization.Serdes;
import org.springframework.kafka.support.serializer.JsonDeserializer;
import org.springframework.kafka.support.serializer.JsonSerializer;

public class AgregateTotalDtoSerDes extends Serdes.WrapperSerde<AggregateTotalDto>{
   public AgregateTotalDtoSerDes() {
      super(new JsonSerializer<>(), new JsonDeserializer<>(AggregateTotalDto.class));
   }
}
```

h. La classe KafkaProducerConfig

```java
package ma.formations.kafka.streams.config;

import ma.formations.kafka.streams.dtos.VentesDto;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.support.serializer.JsonSerializer;

import java.util.HashMap;
import java.util.Map;


@Configuration
public class KafkaProducerConfig {

   @Value(value ="${spring.kafka.bootstrap-server}")
   private String bootstrapAddress;

  @Bean
   public KafkaTemplate<String, VentesDto> kafkaTemplate() {
     Map<String, Object> props = new HashMap<>();
     props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);
     props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
     props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class);
     return new KafkaTemplate<>(new DefaultKafkaProducerFactory<>(props));
   }
}
```

i. La classe KafkaConsumerConfig

```java
package ma.formations.kafka.streams.config;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;

import java.util.HashMap;
import java.util.Map;

@Configuration
public class KafkaConsumerConfig {
```

```
    @Value(value = "${spring.kafka.bootstrap-server}")
    private String bootstrapAddress;

    @Bean
    public ConsumerFactory<String, String> consumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "demo-1");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        return new DefaultKafkaConsumerFactory<>(props);
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory = new
ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        return factory;
    }
}
```

j. La classe KafkaStreamsConfig

```
package ma.formations.kafka.streams.config;

import ma.formations.kafka.streams.dtos.VentesDto;
import ma.formations.kafka.streams.service.KStreamProcessor;
import ma.formations.kafka.streams.service.KTableProcessor;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.kstream.Consumed;
import org.apache.kafka.streams.kstream.KStream;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.annotation.EnableKafka;
import org.springframework.kafka.annotation.EnableKafkaStreams;
import org.springframework.kafka.annotation.KafkaStreamsDefaultConfiguration;
import org.springframework.kafka.config.KafkaStreamsConfiguration;

import java.util.HashMap;
import java.util.Map;

import static org.apache.kafka.streams.StreamsConfig.*;
```

```
@EnableKafka
@EnableKafkaStreams
@Configuration
public class KafkaStreamsConfig {
   @Value(value = "${spring.kafka.bootstrap-server}")
   private String bootstrapAddress;
   @Value(value = "${spring.kafka.topic.input}")
   private String inputTopic;
   @Autowired
   private KStreamProcessor kstreamProcessor;
   @Autowired
   private KTableProcessor ktableProcessor;

   @Bean(name = KafkaStreamsDefaultConfiguration.DEFAULT_STREAMS_CONFIG_BEAN_NAME)
   public KafkaStreamsConfiguration kStreamsConfig() {
      Map<String, Object> props = new HashMap<>();
      props.put(APPLICATION_ID_CONFIG, "streams-app");
      props.put(BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);
      props.put(DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
      props.put(DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.Double().getClass().getName());
      props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
      return new KafkaStreamsConfiguration(props);
   }
   @Bean
   public KStream<String, VentesDto> kStream(StreamsBuilder kStreamBuilder) {
      KStream<String, VentesDto> stream = kStreamBuilder.stream(inputTopic, Consumed.with(Serdes.String(),
new VentesDtoSerDes()));
      //Process KStream
      this.kstreamProcessor.process(stream);
      //Process KTable
      this.ktableProcessor.process(stream);
      return stream;
   }
}
```

k. La classe KStreamProcessor

```
package ma.formations.kafka.streams.service;

import ma.formations.kafka.streams.dtos.VentesDto;
import org.apache.kafka.streams.kstream.KStream;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class KStreamProcessor {
   @Value("${spring.kafka.topic.output}")
   private String outputTopic;
   public void process(KStream<String, VentesDto> stream){
      stream.filter((key, object) -> {
         return object != null && object.getCity() != null && object.getCity().trim().equalsIgnoreCase("casa");
```

```
        }).to(outputTopic);

    }
}
```

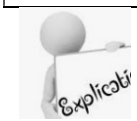
I. La classe KTableProcessor

```
package ma.formations.kafka.streams.service;

import ma.formations.kafka.streams.config.AgregateTotalDtoSerDes;
import ma.formations.kafka.streams.dtos.AggregateTotalDto;
import ma.formations.kafka.streams.dtos.VentesDto;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KeyValue;
import org.apache.kafka.streams.kstream.KGroupedStream;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;
import org.apache.kafka.streams.kstream.Materialized;
import org.apache.kafka.streams.state.KeyValueBytesStoreSupplier;
import org.apache.kafka.streams.state.Stores;
import org.springframework.stereotype.Component;

@Component
public class KTableProcessor {

    public void process(KStream<String, VentesDto> stream){
        KeyValueBytesStoreSupplier customerSales = Stores.persistentKeyValueStore("customer-sales-amount");
        KGroupedStream<String, Double> salesByCustomer = stream
                .map((key, sales) -> new KeyValue(sales.getCustomerCode(), Double.parseDouble(sales.getPrice())))
                .groupByKey();
        KTable<String, AggregateTotalDto> customerAggregate = salesByCustomer.aggregate(() -> new
AggregateTotalDto(),
                (k,v,aggregate) -> {
                    aggregate.setCount(aggregate.getCount()+1);
                    aggregate.setAmount(aggregate.getAmount()+v);
                    return aggregate;
                }, Materialized.with(Serdes.String(),new AgregateTotalDtoSerDes()));
        final KTable<String, Double> dealerTotal =
                customerAggregate.mapValues(value -> value.getAmount(),Materialized.as(customerSales));
    }
}
```



```
KeyValueBytesStoreSupplier customerSales = Stores.persistent
```

- Un store persistant nommé "customer-sales-amount" est créé pour conserver les données d'état (montants totaux par client).
- Les données sont enregistrées localement et sauvegardées dans Kafka, ce qui garantit leur tolérance aux pannes.

```
KGroupedStream<String, Double> salesByCustomer = stream
    .map((key, sales) -> new KeyValue(sales.getCustomerCode(
```

- La méthode map remplace chaque paire clé-valeur (key, sales) par une nouvelle paire (sales.getCustomerCode(), sales.getPrice()).
- Les paires clé-valeur sont regroupées par clé (le code client) pour préparer une agrégation.

```
KTable<String, AggregateTotalDto> customerAggregate = 
    () -> new AggregateTotalDto(), // Initialisateur de
    (k, v, aggregate) -> {
        aggregate.setCount(aggregate.getCount() + 1); 
        aggregate.setAmount(aggregate.getAmount() + v);
        return aggregate;
```

- ***Initialisation*** : L'agrégation commence avec une valeur initiale de type AggregateTotalDto.
- ***Logic d'agrégation*** :
  - ✓ Pour chaque nouvelle valeur v (le montant de la vente), le total du montant (aggregate.getAmount()) est mis à jour.
  - ✓ Le compteur des ventes (aggregate.getCount()) est incrémenté.
- ***Matérialisation*** : Le résultat est stocké dans un **KTable** avec une sérialisation personnalisée pour l'objet AggregateTotalDto.

```
final KTable<String, Double> dealerTotal = customer
    value -> value.getAmount(),
    Materialized.as(customerSales)
```

- A partir de la table customerAggregate, seules les valeurs des montants totaux (value.getAmount()) sont extraites.
- Le résultat est une nouvelle KTable (dealerTotal) contenant les montants totaux par client.
- Les données de dealerTotal sont matérialisées dans le KeyValue Store persistant "customer-sales-amount", ce qui les rend disponibles pour des requêtes ou une récupération ultérieure.

m. La classe KafkaCasaTopicListener

```
package ma.formations.kafka.streams.service;

import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.messaging.handler.annotation.Payload;
import org.springframework.stereotype.Component;

@Component
public class KafkaCasaTopicListener {
   @KafkaListener(topics = "${spring.kafka.topic.output}")
   public void readRxClaimStream(@Payload String record) {
      if(record!=null && record.length()>0) {
         try {
            System.out.println("CASA TOPIC => " + record);
         }catch(Exception e) {
            e.printStackTrace();
         }
      }
   }
}
```

n. La classe StreamsController

```
package ma.formations.kafka.streams.controllers;

import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StoreQueryParameters;
import org.apache.kafka.streams.state.QueryableStoreTypes;
import org.apache.kafka.streams.state.ReadOnlyKeyValueStore;
import org.springframework.kafka.config.StreamsBuilderFactoryBean;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/ventes")
```

```
public class StreamsController {
   private final StreamsBuilderFactoryBean factoryBean;

   public StreamsController(StreamsBuilderFactoryBean factoryBean) {
      this.factoryBean=factoryBean;
   }

   @GetMapping("/customer/{id}")
   public String getDealerSales(@PathVariable String id){
      KafkaStreams kafkaStreams =  factoryBean.getKafkaStreams();
      ReadOnlyKeyValueStore<String, Long> amounts = kafkaStreams
            .store(StoreQueryParameters.fromNameAndType("customer-sales-amount",
QueryableStoreTypes.keyValueStore()));
      return "Total Article Sales for Customer "+id +" is MAD"+ amounts.get(id);
   }
}
```

o. La classe KafkaProducerController

```
package ma.formations.kafka.streams.controllers;

import ma.formations.kafka.streams.dtos.VentesDto;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;
import java.util.Random;
import java.util.UUID;


@RestController
public class KafkaProducerController {
   @Value(value = "${spring.kafka.topic.input}")
   private String inputTopic;
   private KafkaTemplate<String, VentesDto> kafkaTemplate;
   Random random = new Random();
   private static final List<String> types=List.of("ELECTROMENAGER","INFORMATIQUE","Divers");

   public KafkaProducerController(KafkaTemplate<String, VentesDto> kafkaTemplate) {
      this.kafkaTemplate = kafkaTemplate;
   }
   @GetMapping("/send/{customer}/{city}/{price}")
   public String sendMessage(@PathVariable String customer,@PathVariable String city,@PathVariable String
price) {
      kafkaTemplate.send(inputTopic, VentesDto.builder().
                  customerCode(customer).
                  city(city).
                  price(price).
```

```
                transactionId(UUID.randomUUID().toString()).
                typeArticle(types.get(random.nextInt(types.size()))).
                build()
        );
    return "Messages sent with success ";
  }
}
```

p. La classe MainApplication

```
package ma.formations.kafka.streams;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MainApplication {

  public static void main(String[] args) {
    SpringApplication.run(MainApplication.class, args);
  }

}
```

VI. Tests

- Démarrer Kafka :

```
docker run -p 9092:9092 apache/kafka:3.9.0
```

- Créer les deux topic "input-ventes" et "casa-ventes" :

sh kafka-topics.sh --bootstrap-server localhost:9092 --create --topic input-ventes --partitions 1 --replication-factor 1

sh kafka-topics.sh --bootstrap-server localhost:9092 --create --topic casa-ventes --partitions 1 --replication-factor 1

- Lancer la méthode main de la classe MainApplication.
- Publier quelques messages en exécutant les URLs suivants, par exemple :

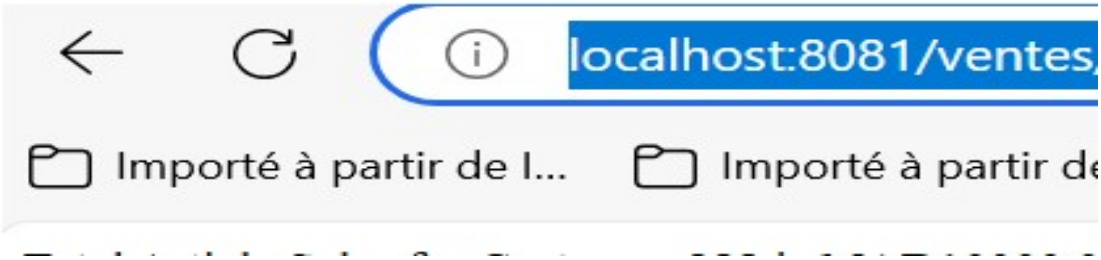
http://localhost:8081/send/888/casa/1000
http://localhost:8081/send/222/casa/4000

http://localhost:8081/send/888/casa/3000
http://localhost:8081/send/222/rabat/1000
http://localhost:8081/send/888/casa/6000

- Vérifier que le Listner que vous avez développé a bien fonctionné :

```
CASA TOPIC => {"city":"casa","customerCode":"111","price":"1000","transactionId":"b56007e2-bb29-4
CASA TOPIC => {"city":"casa","customerCode":"111","price":"4000","transactionId":"fc36ad8b-719c-4
2025-01-06T19:17:51.295+01:00  INFO 23556 --- [read-1-producer] org.apache.kafka.clients.Metadata
2025-01-06T19:17:51.329+01:00  INFO 23556 --- [read-1-producer] org.apache.kafka.clients.Metadata
```

- Exécuter le lien http://localhost:8081/ventes/customer/888 et vérifier que le résultat est bien 10000 DH



- Exécuter le lien http://localhost:8081/ventes/customer/222 et vérifier que le résultat est bien 5000 DH