Apache Kafka - Conceptos básicos

Broker

Un servidor de kafka es conocido como un Broker, un cluster de kafka consiste en múltiples brokers.

Mensaje

Un mensaje es una unidad de datos en Kafka, está compuesto por una llave y un valor.

Topics / Partitions / Offsets

Un **topic** es un stream (flujo) de mensajes, cada topic se compone de una o más **particiones.** Los mensajes son colocados en las particiones y son ordenados de acuerdo a un número llamado **offset**.

Tener múltiples particiones permite tener más consumidores de mensajes funcionando de forma concurrente, esto se ve reflejado en el incremento del throughput(tasa de transferencia).

Topics / Partitions / Offsets

Un **topic** es un stream (flujo) de mensajes, cada topic se compone de una o más **particiones.** Los mensajes son colocados en las particiones y son ordenados de acuerdo a un número llamado **offset**.

Tener múltiples particiones permite tener más consumidores de mensajes funcionando de forma concurrente, esto se ve reflejado en el incremento del throughput(tasa de transferencia).

Replicas

Para incrementar la disponibilidad de la información, los topics se deben replicar en múltiples brokers, esto se define en el atributo **replication-factor**, el cual define el número de copias de la información. Cada topic tendrá asignado un líder y seguidores.

Una replica actualizada se conocerá como (**In-sync**) y se mantiene actualizada con los cambios del líder. Si el líder falla una In-sync se puede convertir en lider.

En un ambiente saludable todas las replicas deben encontrarse In-sync, es aceptable que no se encuentren de ese modo después de un fallo.

Producer

Un producer es un componente que publica mensajes en uno o más topics. Puede seleccionar en que partición desea colocar los mensajes

Consumer

Un consumer lee mensajes de uno o más topics y los procesa. La diferencia entre el la posición actual del consumer y el mensaje más nuevo de la partición se conoce como (**offset lag**).

Si el offset lag no es muy largo eso no es un problema, se puede convertir en uno si se acerca al ***rentention period*** (Periodo de tiempo que se mantienen los mensajes en kafka).

Consumer groups

Un consumer group contiene uno o más consumidores de mensajes, Los mensajes de una partición son procesados por un consumidor del grupo (Esto asegura que los mensajes se procesen en orden).

Si hay más particiones que consumers, un consumer puede leer más de una partición. Si Hay más consumers que particiones algunos se quedarán sin partición a consumir.

Descarga kafka

Para descargar apache kafka debes acceder a la siguiente url :

https://kafka.apache.org/downloads

Asegurate de descargar la versión que dice binary downloads.

Inicia kafka

Para iniciar tu servidor de kafka deberás ejecutar los siguientes comandos:

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

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

Esto iniciará tanto zookeeper como kafka.

Creando un topic

Los mensajes se procesan en topics, para crear uno deberás ejecutar el siguiente comando:

```
$ bin/kafka-topics.sh
--bootstrap-server
localhost:9092 --create --topic
devs4j-topic  --partitions 5
--replication-factor 1
```

Este comando recibe los siguientes parámetros:

- `bootstrap-server` = Kafka server
- `topic` = Nombre del topic a crear
- `partitions` = Número de particiones
- `replication-factor` = Número de réplicas por broker

Listando topics

Puedes listar los topics disponibles ejecutando:

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

Salida de ejemplo:

```
devs4j-topic
```

Ver definición de un topic

Si deseas consultar como se definió un topic puedes describirlo con el siguiente comando:

```
$ bin/kafka-topics.sh --describe
--topic devs4j-topic
--bootstrap-server
localhost:9092
```

Salida de ejemplo:

```
Topic: devs4j-topic        PartitionCount: 5
        ReplicationFactor: 1

        Topic: devs4j-topic        Partition: 0  Leader: 0
        Replicas: 0  Isr: 0
        Topic: devs4j-topic        Partition: 1  Leader: 0
        Replicas: 0  Isr: 0
        Topic: devs4j-topic        Partition: 2  Leader: 0
        Replicas: 0  Isr: 0
        Topic: devs4j-topic        Partition: 3  Leader: 0
        Replicas: 0  Isr: 0
        Topic: devs4j-topic        Partition: 4  Leader: 0
        Replicas: 0  Isr: 0
```

De la salida podemos observar lo siguiente:

- Tiene 5 particiones, que
- Solo se tiene una replica
- Solo hay un líder el cual es quien contiene el topic especificado.

Crear un producer

Para iniciar un producer ejecutaremos el siguiente comando:

```
$ bin/kafka-console-producer.sh --topic
devs4j-topic --bootstrap-server
localhost:9092
```

Crear un consumer

Para iniciar un consumer ejecutaremos el siguiente comando:

```
$ bin/kafka-console-consumer.sh --topic
devs4j-topic --from-beginning
--bootstrap-server localhost:9092

--property print.key=true --property
key.separator="-"
```

El parámetro `--from-beginning` permite especificar si queremos recibir solo los mensajes nuevos o queremos leer todos desde el inicio.

Modificando un topic

Para modificar un topic existente ejecutaremos el siguiente comando:

```
$ bin/kafka-topics.sh --bootstrap-server
localhost:9092 --alter --topic devs4j-topic \
--partitions 40
```

En el ejemplo anterior se cambia el número de particiones a 40.

Agregando configuraciones

Para modificar las configuraciones de un topic existente ejecutaremos el siguiente comando:

```
$ bin/kafka-configs.sh --bootstrap-server
localhost:9092 --entity-type topics
--entity-name devs4j-topic --alter
--add-config x=y
```

Quitando configuraciones

Para remover las configuraciones de un topic existente ejecutaremos el siguiente comando:

```
$ bin/kafka-configs.sh --bootstrap-server
localhost:9092 --entity-type topics
--entity-name devs4j-topic --alter
--delete-config x
```

Borrando un topic

Para borrar un topic existente ejecutaremos el siguiente comando:

```
$bin/kafka-topics.sh --bootstrap-server
localhost:9092 --delete --topic devs4j-topic
```

Kafka no permite reducir el número de particiones.

Para limpiar el ambiente de los ejemplos de prueba terminaremos los procesos en el siguiente orden:

- Control + C en los consumers y producers
- Control + C en los brokers de kafka
- Control + C en el servidor de zookeeper

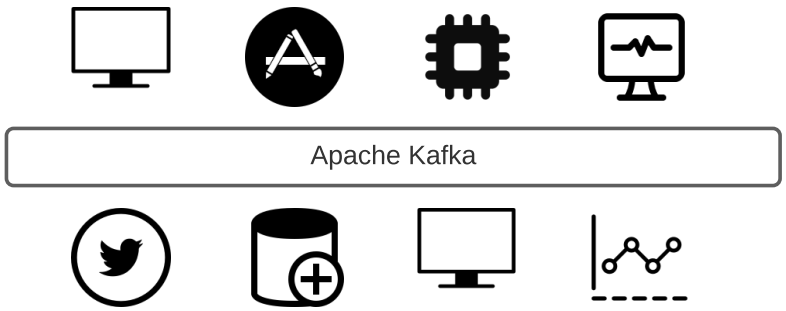
Si se desea borrar la información, ejecutar:

```
$ rm -rf /tmp/kafka-logs /tmp/zookeeper
```

DEVS4J
Hecho por Devs para Devs

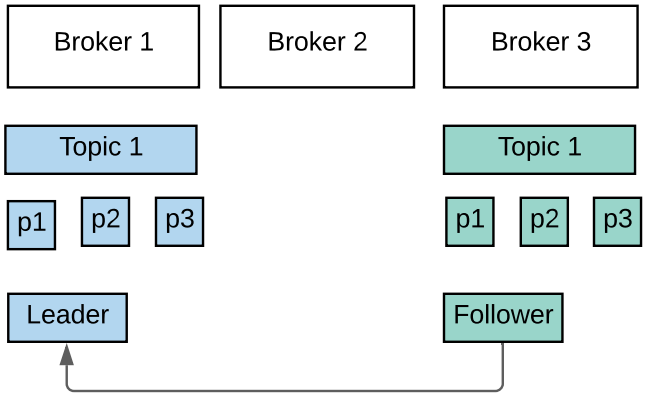
Apache Kafka - Diagramas

Kafka use cases

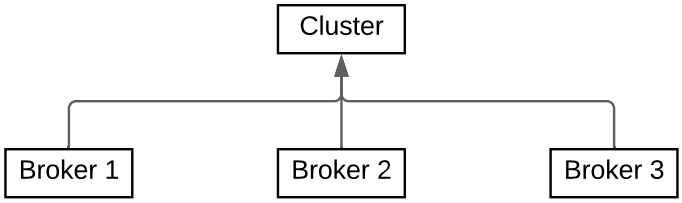


Replicas

Replication factor =2



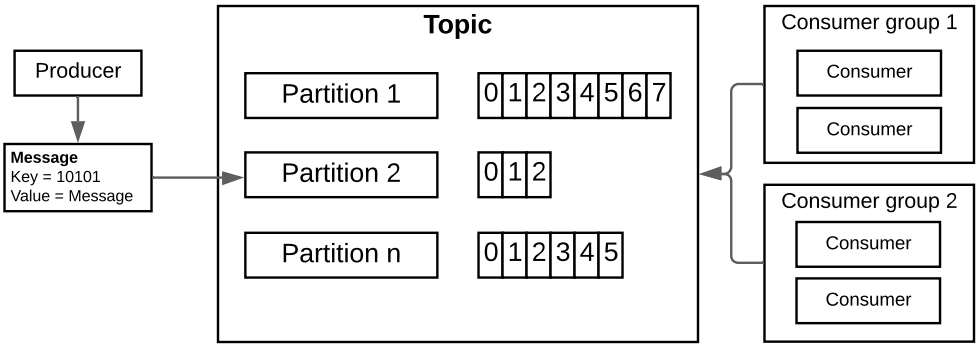
Kafka Cluster



Estructura de un mensaje

Key = 10101
Value = Message
Timestamp = 10/08/2020T00:00:00

Flujo de entrega de mensajes



DEVS4J
Hecho por Devs para Devs

Apache Kafka - Procesamiento de mensajes

Configuración

Para enviar mensajes es necesario agregar las siguientes dependencias:

```
<dependencies>
      <dependency>
            <groupId>org.apache.kafka</groupId>
             <artifactId>kafka-clients</artifactId>
             <version>2.3.0</version>
      </dependency>
      <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>1.6.1</version>
      </dependency>
</dependencies>
```

De igual modo incluiremos el siguiente pluging para especificar la versión de Java.

```
<build>
 <plugins>
  <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
      <source>1.8</source>
      <target>1.8</target>
      </configuration>
  </plugin>
 </plugins>
</build>
```

Java Producer

El código necesario para enviar mensajes con Java es el siguiente:

```
Properties props=new Properties();
props.put("bootstrap.servers","localhost:9092");
props.put("acks","1");
props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

try(Producer<String, String>producer=new
KafkaProducer<>(props);) {
      for(inti= 0; i< 100 ;i++) {
            producer.send(new
      ProducerRecord<>("devs4j-topic","key","message"));
      }
      producer.flush();
}
```

Producer properties

A continuación se listan algunas propiedades importantes a configurar en los producers:

- bootstrap.servers - Server + Puerto de todos los brokers del cluster

- acks - Define si se requiere un ack (confirmación de recibido) del mensaje, los valores posibles son 0 (no), 1 (si, de un broker), all (todos los brokers).

- key.serializer - Define la clase Java que va a serializar la llave del mensaje.

- value.serializer - Define la clase Java que va a serializar el mensaje.

-linger.ms -Los batches son agrupados por tiempo, este tiempo esta determinado en milisegundos (Muy relevante haciendo pruebas de performance).

-batch.size - Los mensajes son enviados en batches para mejorar el performance.

-buffer.memory - Define el tamaño máximo del buffer en el que se pueden colocat batches,

Java Consumer

El código para consumir mensajes desde Java es el siguiente:

```
Properties props=new Properties();
props.setProperty("bootstrap.servers","localhost:9092");
props.setProperty("group.id","devs4j-group");
props.setProperty("enable.auto.commit","true");
props.setProperty("auto.commit.interval.ms","1000");
props.setProperty("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
props.setProperty("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");


try(KafkaConsumer<String, String> consumer=new
KafkaConsumer<>(props);) {
      consumer.subscribe(Arrays.asList("devs4j-topic"));
      while(true) {
            ConsumerRecords<String,String>records=
            consumer.poll(Duration.ofMillis(100));
            for(ConsumerRecord<String, String>record
:records)
            log.info("offset = {}, key = {}, value ={}",
            record.offset(),record.key(),record.value());
      }
}
```

Consumer properties

A continuación se listan algunas propiedades importantes a configurar en los consumers:

- bootstrap.servers - Server + Puerto de todos los brokers del cluster

- group.id - Identificador único que determina el consumer group al que pertenece el consumer.

- enable.auto.commit - Si es verdadero se realizará commit a los offsets de forma periodica en segundo plano.

- auto.commit.interval.ms: La frecuencia en milisegundos en que se realizará commit a los offsets (enable.auto.commit debe ser true)

- key.deserializer - Define la clase Java que va a deserializar la llave.

-value.deserializer -Define la clase Java que va a deserializar el mensaje.

Transactional

Es posible manejar transacciones utilizando kafka, para esto se deben seguir los siguientes puntos:

- Producer
      -Definir acks =all
      -Asignar un transactional id
      transactional.id=devs4j-producer

- Consumer
      -Definir isolation.level=read_committed

Si hay algún error los mensajes se encontrarán en el topic pero como read_uncommitted.

Si no hay ningún error, los mensajes en el topic pasarán de read_uncommitted a read_committed y serán leidos por el consumer.

A demás de las configuraciones anteriores se deben ejecutar los siguientes métodos del producer:

- producer.initTransactions();
- producer.beginTransaction();
- producer.commitTransaction();
- producer.abortTransaction();

Producer transaccional

A continuación las modificaciones para convertir el producer en transaccional:

```
Properties props=new Properties();
props.put("bootstrap.servers","localhost:9092");
props.put("acks","all");
props.put("compression.type","gzip");
props.put("linger.ms","1");
props.put("batch.size","32384");
props.put("transactional.id","devs4j-producer");
props.put("buffer.memory","33554432");
props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

try(Producer<String, String> producer = new
KafkaProducer<>(props)) {
 try{
  producer.initTransactions();
  producer.beginTransaction();

  for(inti= 0;i< 100000;i++) {
       producer.send(new
       ProducerRecord<>("devs4j-topic","message"));
       if(i== 100) {
            throw new Exception("Random exception");
       }
 }
  producer.commitTransaction();
 }catch(Exception e) {
       producer.abortTransaction();
       log.error("Error processing messages",e);
 }
}
```

Consumer transaccional

A continuación las modificaciones para convertir el consumer en transaccional:

```
Properties props=new Properties();
props.setProperty("bootstrap.servers","localhost:9092");
props.setProperty("group.id","devs4j-group");
props.setProperty("enable.auto.commit","true");
props.setProperty("auto.commit.interval.ms","1000");
props.setProperty("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
props.setProperty("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");


props.setProperty("isolation.level","read_committed");


try(KafkaConsumer<String, String>consumer=new
KafkaConsumer<>(props);) {
      consumer.subscribe(Arrays.asList("devs4j-topic"));
      while(true) {
            ConsumerRecords<String,String>records=
            consumer.poll(Duration.ofMillis(100));
            for(ConsumerRecord<String,
                  String>record:records)
                  log.info("partition = {} , offset = {}, key = {},
                  value = {}",record.partition(),record.offset(),
                  record.key(),record.value());
            }
}
```

DEVS4J
Hecho por Devs para Devs

www.twitter.com/devs4j

www.facebook.com/devs4j

www.devs4j.com

Apache Kafka - Conceptos avanzados

Callbacks

Los callbacks son utilizados en los producers para notificar el status del envío de un mensaje, a cotninuación un ejemplo:

Utilizando clases anónimas:

```
producer.send(new
ProducerRecord<>("devs4j-topic","message"), new
Callback() {
    @Override
    public void onCompletion(RecordMetadata
    metadata, Exceptionexception) {
        if(exception!=null) {
            log.error("Error sending the message"
            ,exception);
        }
        log.info("Partition = {} Offset ={}",
        metadata.partition(),metadata.offset());
 }
});
```

Utilizando lambdas:

```
producer.send(new
ProducerRecord<>("devs4j-topic","message"),
(metadata,exception) -> {
    if(exception!=null) {
        log.error("Error sending the message ",
        exception);
    }
    log.info("Partition = {} Offset ={}",
    metadata.partition(),metadata.offset());
});
```

Asegurar el orden

Cuando se tiene un ambiente multithread, es posible perder el orden de los mensajes, en algunos casos esto esta bien pero en otros no puede suceder.

En casos en los que se deba preservar el orden de los mensajes se debe hacer uso de message keys.

Ejercicio con producers

Enviar a un topic con 5 particiones mensajes con la siguiente estructura:

Usuario	Acción	Monto	Timestamp
1020	Deposito	200	2020-08-25T00:00
1020	Deposito	100	2020-08-25T01:00
1020	Deposito	200	2020-08-25T02:00
1020	Retiro	-300	2020-08-25T03:00
1021	Deposito	200	2020-08-25T00:00
1021	Deposito	200	2020-08-25T00:00

Cada registro será una transacción y será un mensaje en kafka, como se puede ver en el caso de ejemplo el orden se debe preservar dado que no se puede aplicar un retiro sin tener un deposito previo.

Detalles de implementación

Se recomienda utilizar el usuario como message key para preservar el orden de los mensajes a entregar.

Ejercicio consumers

Inicia más de un consumer con el mismo consumer group y revisa que los mensajes se lean sin que se pierda el orden en el que se enviaron, no debe importar la cantidad de consumers.

El problema del ordenamiento

Podemos asegurar el **orden** en el que se envían los mensajes a través de **message keys,** asegurando que los mensajes con las mismas llaves vayan a la misma partición, pero en caso de que exista un reintento (la configuración `retries > 0`) al enviar un mensaje, existe la posibilidad de que se vaya a la partición correcta pero se pierda el orden.

Debido a lo anterior, si es totalmente necesario que se preserve el orden de los mensajes, es recomendable que se configure **max.in.flight.requests.per.connection** = 1. Esto hará que solo se pueda realizar una petición de envío de mensajes al mismo tiempo (su valor por defecto es 5).

Es importante aclarar que si colocamos la propiedad **max.in.flight.requests.per.connection** = 1 el performance o throughput del producer se verá impactado.

Multi thread consumer

Para consumir mensajes de kafka a través de múltiples threads escribiremos el siguiente código:

```
public class Devs4jThreadConsumer extends Thread {
    private final KafkaConsumer<String,
    String>consumer;

    private final AtomicBoolean closed = new
    AtomicBoolean(false);

    public static final Logger log=
    LoggerFactory.getLogger(Devs4jThreadConsumer
    .class);

    public Devs4jThreadConsumer
    (KafkaConsumer<String, String>consumer) {
        this.consumer=consumer;
    }

    @Override
    public void run() {
        try{
    consumer.subscribe(Arrays.asList("devs4j-topic"));
    while(!closed.get()) {
        ConsumerRecords<String, String>records=
        consumer.poll(Duration.ofMillis(10000));
        for(ConsumerRecord<String, String>record
        :records) {
            log.info("offset = {}, key = {}, value =
        {}",record.offset(),record.key(),record.value());
        }
    }
 }catch(WakeupException e) {
    if(!closed.get())
        throw e;
 }finally{
    consumer.close();
 }
}
    public void shutdown() {
        closed.set(true);
        consumer.wakeup();
    }
}
```

Idempotent producers

Si se desea asegurar que no se generan mensajes duplicados en kafka se debe colocar la propiedad **enable.idempotence =true.**

enable.idempotence requiere que:

- max.in.flight.requests.per.connection < = 5

- retries > 0

- acks > all

Si algúno de ellos no es correcto, se producirá una ConfigException.

Ejecutando consumer threads

Para ejecutar los threads definidos anteriormente ejecutaremos el siguiente código:

```
public class Devs4jMultithreadConsumer {
    public static void main(String[]args) {
      //Config properties
    ExecutorService executor=
    Executors.newFixedThreadPool(5);
     for(inti= 0;i< 5;i++) {
        Runnable worker= new
    Devs4jThreadConsumer(newKafkaConsumer<>(props));
        executor.execute(worker);
    }
    executor.shutdown();
    while(!executor.isTerminated()) ;
    }
}
```

Assign

El método **assign**(Collection<TopicPartition> partitions) recibe como parámetro una lista de particiones y las asigna al consumer. Si se ejecuta más de una vez no agregará una partición sino que reemplazará la lista de particiones, a continuación un ejemplo:

```
TopicPartition topicPartition= new TopicPartition
("devs4j-topic", 0);
consumer.assign(Arrays.asList(topicPartition));
```

Seek

El método **seek**(TopicPartition partition, long offset) sobreescribe el offset que el consumer utilizará cuando se ejecute el métod **poll**(timeout), de este modo podemos volver al pasado y leer mensajes que ya habían sido leidos por nuestro consumer, a continuación un ejemplo:

```
try(KafkaConsumer<String, String>consumer=new
KafkaConsumer<>(props);) {
    TopicPartition topicPartition=new
    TopicPartition("devs4j-topic", 0);

    consumer.assign(Arrays.asList(topicPartition));
    consumer.seek(topicPartition, 12);

    while(true) {
        ConsumerRecords<String, String>records=
        consumer.poll(Duration.ofMillis(100));
        for(ConsumerRecord<String,
    String>record:records)
            log.info("partition = {} , offset = {}, key = {},
            value = {}",record.partition(),record.offset(),
            record.key(),record.value());
    }
}
```

Disable auto commit

Cuando definimos enable.auto.commit =true (Opción por defecto) procesamos los mensajes de forma síncrona pero se hace commit de los offsets de forma asíncrona, podemos deshabilitarlo y ejecutar el método del consumer **commitSync**();.

DEVS4J
Hecho por Devs para Devs

www.twitter.com/devs4j

www.facebook.com/devs4j

www.devs4j.com

Apache Kafka - Integración con Spring

Configuración

Para configurar Spring kafka se debe incluir la siguiente dependencia:

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

Consumer properties

A continuación se listan las configuraciones de un consumer de kafka:

```
@Bean
public Map<String, Object> consumerProps() {
        Map<String, Object>props=new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
        "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
        "group");
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
        true);
        props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG,
        "100");
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG,
        "15000");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
        IntegerDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        StringDeserializer.class);
        return props;
 }
```

A continuación la explicación de cada una:

- ***BOOTSTRAP_SERVERS_CONFIG :*** Lista de brokers de kafka en el cluster.
- ***GROUP_ID_CONFIG:*** Consumer group que consumirá los mensajes
- ***ENABLE_AUTO_COMMIT_CONFIG:*** Determina si se hará commit al offset de forma periódica
- ***AUTO_COMMIT_INTERVAL_MS_CONFIG:*** Determina la frecuencia en milisegundos en la que se hará commit a los offsets, solo es necesaria si ***ENABLE_AUTO_COMMIT_CONFIG =true.***
- ***SESSION_TIMEOUT_MS_CONFIG:*** Timeout utilizado para determinar errores en los clientes.
- ***KEY_DESERIALIZER_CLASS_CONFIG:*** Clase a utilizar para deserializar la llave
- ***VALUE_DESERIALIZER_CLASS_CONFIG:*** Clase a utilizar para deserializar el mensaje

Lectura de mensajes de kafka

Una vez definidas las propiedades se debe configurar el listener:

```
@Bean
public ConsumerFactory<Integer, String> consumerFactory() {
    return new
DefaultKafkaConsumerFactory<>(consumerProps());
}

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

Para utilizar el listener crearemos el siguiente método en un componente de spring

```
@KafkaListener(topics ="devs4j-topic", groupId ="consumer")
public void listen(String message) {
     System.out.println("Received Messasge in group foo:
"+message);
}
```

Producer properties

A continuación se listan las configuraciones de un producer de kafka:

```
private Map<String, Object> producerProps() {
        Map<String, Object> props=new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
        "localhost:9092");
        props.put(ProducerConfig.RETRIES_CONFIG, 0);
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);
        props.put(ProducerConfig.LINGER_MS_CONFIG, 1);
        props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33554432);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
        IntegerSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        StringSerializer.class);
        return props;
}

@Bean
 public KafkaTemplate<Integer, String> createTemplate() {
        Map<String, Object>senderProps= producerProps();
        ProducerFactory<Integer, String> pf= new
        DefaultKafkaProducerFactory<Integer, String>(senderProps);
        KafkaTemplate<Integer, String> template=new KafkaTemplate<>(pf);
        return template;
 }
```

Se incluyen las siguientes nuevas propiedades:

- ***RETRIES_CONFIG:***Define los reintentos que se realizarán en caso de error.
- ***BATCH_SIZE_CONFIG:***El producer agrupará los registros en batches, mejorando el performance (está definido en bytes).
- ***LINGER_MS_CONFIG:*** Los batches se agruparan de acuerdo de un periodo de tiempo, está definido en milisegundos.
- ***BUFFER_MEMORY_CONFIG:*** Define el espacio de memoria que se asignará para colocar los mensajes que están pendientes por enviar.

Utilizando el producer

Una vez configurado el kafka template lo utilizaremos como se muestra a continuación:

```
@Autowired
 private KafkaTemplate<Integer, String> kafkaTemplate;


public Role createRole(Rolerole) {
        Role save = repository.save(role);
        kafkaTemplate.send("devs4j-topic",save.getName());
        return save;
 }
```

El código anterior publicará un mensaje en el topic devs4j-topic con un message key igual al id del objeto y un mesaje con el contenido del objeto.

Callbacks con Spring Async

A continuación se muestra como realizar callbacks utilizando Spring:

```
ListenableFuture<SendResult<String, String>>future=
kafkaTemplate.send("devs4j-topic","Sample message ");

future.addCallback(new KafkaSendCallback<String, String>() {

    @Override
    public void onSuccess(SendResult<String, String>result) {
        log.info("Message sent");
    }

    @Override
    public void onFailure(Throwable ex) {
        log.error("Error sending message ",ex);
    }

    @Override
    public void onFailure(KafkaProducerException ex) {
        log.error("Error sending message ",ex);
    }
});
```

Envío de mensajes síncrono

A continuación se muestra como enviar mensajes de forma síncrona:

```
kafkaTemplate.send(new
ProducerRecord<String,String>
("key","value")).get();

kafkaTemplate.send(new
ProducerRecord<String,String>
("key","value")).get(10,TimeUnit.SECONDS);
```

En el segundo ejemplo vemos como podemos definir un timeout para esperar y leer el resultado si está presente, si no es así se producirá una TimeoutException.

Batch listeners

Es posible leer más de un registro al mismo tiempo, para esto agregaremos la siguiente configuración al **ConcurrentKafkaListenerContainerFactory**

```
listenerContainerFactory.
setBatchListener(true);
```

Además de la configuración anterior agregaremos algunos parámetros a nuestra anotación **@KafkaListener** como se muestra a continuación:

```
@KafkaListener(topics ="devs4j-topic",
  containerFactory
="listenerContainerFactory",groupId
="devs4j-group3", properties =
{"max.poll.interval.ms:60000",
  "max.poll.records:100"})
```

A continuación los puntos importantes sobre los atributos:

- **containerFactory** : Es el nombre del método que crea el bean **ConcurrentKafkaListenerContainerFactory**

- **properties**: Permite definir propiedades del consimidor.

A continuación el significado de las propiedades definidas:

- max.poll.interval.ms: Define el tiempo entre una ejecución y otra para el método pool.

- max.poll.records: Define el máximo número de registros a devolver por el método pool.

Por último se debe modificar el método al que se aplica la anotación para que en lugar de recibir un mensaje de tipo String reciba una lista como se muestra a continuación:

```
public void listen(List<String>messages){
.....
}
```

< DEVS4J >
Hecho por Devs para Devs

www.twitter.com/devs4j

www.facebook.com/devs4j

www.devs4j.com

Apache Kafka - Integración con Spring Pt2

Consumiendo mensajes completos

En el ejemplo anterior se explicó como consumir solo el contenido del mensaje pero no se hablo de como leer información adicional como el offset, la llave y la partición, a continuación se muestra como crear el listener para soportar toda la información:

```
public void listen(List<ConsumerRecord<String,
String>>messages) {
  for(ConsumerRecord<String, String>message :
       messages) {
       log.info("Offset {} Partition= {} Key = {}Value = {}",
       message.offset(),message.
       message.key(),message.value());
    }
}
```

Multithread

Si se desea consumir mensajes de forma concurrente en Spring se debe agregar la siguiente configuración en el **ConcurrentKafkaListenerContainerFactory:**

```
listenerContainerFactory.setConcurrency(3);
```

Con esto se tendrán 3 hilos consumiendo mensajes de forma concurrente.

Los nombres de los hilos se construirán por default del siguiente modo:

- container-0-C-2 - Primer hilo
- container-1-C-2 - Segundo hilo
- container-2-C-2 - Tercer hilo

Habilitando consumer en tiempo de ejecución

En algunos momentos se desea habilitar o deshabilitar un consumer en tiempo de ejecución, para esto haremos lo siguiente:

```
@KafkaListener(id ="autoStartup", autoStartup ="false")
```

- **id**: Define un identificador para nuestro container
- **autoStartup**: Define que nuestro consumer no iniciará por defecto cuando inicie nuestra aplicación.

Para habilitarlo podemos invocar lo siguiente desde cualquier parte de nuestro código:

```
@Autowired
private KafkaListenerEndpointRegistry registry;

public void anyMethod() {
      log.info("Starting consuming messages");
      registry.getListenerContainer("autoStartup").start();
}
```

Uso de métricas

El primer paso para utilizar métricas será incluir la siguiente dependencia en el proyecto:

```
<dependency>
      <groupId>io.micrometer</groupId>
      <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

La dependencia anterior nos proporcionará el soporte para trabajar con métricas utilizando micrometer y prometheus.

Creación de meter registry

Una vez configurada la dependencia de micrometer y prometheus, crearemos un bean de tipo **MetricRegistry** como se muestra a continuación:

```
@Bean
public MeterRegistry meterRegistry() {

 PrometheusMeterRegistry prometheusMeterRegistry=new
PrometheusMeterRegistry (PrometheusConfig.DEFAULT);
 return prometheusMeterRegistry;
}
```

Integración con producer

El siguiente paso será integrar el MeterRegistry en nuestro producer como se muestra a continuación:

```
@Bean
public KafkaTemplate<String, String> kafkaTemplate() {
 DefaultKafkaProducerFactory<String, String>
producerFactory= new DefaultKafkaProducerFactory<>(
   producerProperties());
 producerFactory.addListener(new
MicrometerProducerListener<String,String>
(meterRegistry()));
 KafkaTemplate<String, String> template = new
KafkaTemplate<>(producerFactory);
 return template;
}
```

Lo importante a mencionar en este punto es la inclusión del método meterRegistry() en la creación de un MicrometerProducerListener, quien estará dando seguimiento a las métricas.

Habilitando scheduling

En esta sección utilizaremos la anotación **@Scheduled** así que es necesario que en nuestra clase principal o en la de configuración coloquemos la anotación @EnableScheduling, como se muestra a continuación:

```
@SpringBootApplication
@EnableScheduling
public class CursoKafkaSpringApplication {
      // ...
}
```

Inyectando el MetricRegistry

Una vez configurado inyectaremos nuestro MetricRegistry en nuestra aplicación como se muestra a continuación:

```
@Autowired
 private MeterRegistry meterRegistry;
```

Producer

A continuación un producer que enviará mensajes cada 2 segundos a Kafka:

```
@Scheduled(fixedDelay = 2000, initialDelay = 100)
public void schedule() {
    log.info("Sending messages ");
      for(int i= 0;i< 200;i++) {
            kafkaTemplate.send("devs4j-topic","Message"+i);
      }
}
```

Precaución

Si no hay métricas disponibles se producirá una **MeterNotFoundException**, así que es importante contemplar ese caso durante el desarrollo de nuestras aplicaciones.

Mostrando métricas

```
@Scheduled(fixedDelay = 2000, initialDelay = 500)
public void messageCountMetric() {
      double count =
      meterRegistry.get("kafka.producer.record.send.total")
      .functionCounter().count();
       log.info("Count {} ",count);

}
```

Métricas disponibles

Para conocer las métricas disponibles podremos ejecutar el método MetricRegistry.getMeters() como se muestra a continuación:

```
for(Meter meter:meters) {
      log.info("Metric {} ",meter.getId());
}
```

Algunas métricas disponibles:

```
kafka.producer.outgoing.byte.total
kafka.producer.iotime.total
kafka.producer.buffer.exhausted.total
kafka.producer.io.ratio
kafka.producer.node.request.size.avg
kafka.producer.record.queue.time.max
kafka.producer.successful.authentication.rate
kafka.producer.connection.close.rate
kafka.producer.outgoing.byte.rate
kafka.producer.record.retry.rate
kafka.producer.failed.authentication.total
kafka.producer.node.request.latency.avg
kafka.producer.requests.in.flight
kafka.producer.connection.creation.rate
kafka.producer.network.io.rate
kafka.producer.record.send.total
kafka.producer.produce.throttle.time.max
kafka.producer.metadata.age
kafka.producer.io.wait.ratio
kafka.producer.incoming.byte.rate
kafka.producer.successful.reauthentication.rate
kafka.producer.records.per.request.avg
kafka.producer.request.size.avg
kafka.producer.node.incoming.byte.total
kafka.producer.buffer.available.bytes
kafka.producer.select.rate
kafka.producer.record.retry.total
kafka.producer.reauthentication.latency.max
kafka.producer.produce.throttle.time.avg
kafka.producer.io.waittime.total
kafka.producer.batch.size.avg
kafka.producer.node.response.total
kafka.producer.request.latency.max
kafka.producer.request.size.max
spring.kafka.template
kafka.producer.connection.close.total
kafka.producer.buffer.total.bytes
kafka.producer.node.request.size.max
kafka.producer.request.rate
kafka.producer.record.error.rate
kafka.producer.connection.count
kafka.producer.network.io.total
kafka.producer.node.response.rate
kafka.producer.record.send.rate
```

DEVS4J
Hecho por Devs para Devs

Apache Kafka - ElasticSearch

Iniciando Elasticsearch

Descarga elasticsearch:

```
curl -L -O https://artifacts.elastic.co/
downloads/elasticsearch
/elasticsearch-7.9.0-darwin-x86_64.tar.gz
```

Descomprime el archivo:

```
tar -xvf
elasticsearch-7.9.0-darwin-x86_64.tar.gz
```

Inicia Elasticsearch:

```
cd elasticsearch-7.9.0/bin
./elasticsearch
```

Validando salud

Una vez que iniciaste Elasticsearch el siguiente paso es verificar su salud para esto ejecutaremos el siguiente endpoint:

```
GET /_cat/health?v
```

Indexando documentos

Para indexar documentos ejecutaremos el siguiente endpoint:

PUT /devs4j-transactions/transactions/1

```
{
 "nombre":"Noah",
 "apellido":"Kulas",
 "username":"tristan.lowe",
 "monto":1.9829264751E7
}
```

La petición anterior, creará un índice llamado **devs4j-transactions** y creará un documento con el id 1 que contendrá la información colocada en el json.

Consultando documento por id

Una vez creado el documento, podemos consultarlo ejecutando el siguiente endpoint:

GET /devs4j-transactions/transactions/1

Esto nos devolverá una salida como la siguiente:

```
{
    "_index":"devs4j-transactions",
    "_type":"transactions",
    "_id":"1",
    "_version":1,
    "_seq_no":0,
    "_primary_term":1,
    "found":true,
    "_source": {
        "nombre":"Noah",
        "apellido":"Kulas",
        "username":"tristan.lowe",
        "monto":1.9829264751E7
    }
}
```

Configurando proyecto

Agregaremos la siguiente dependencia:

```
<dependency>
  <groupId>org.elasticsearch.client</groupId>
  <artifactId>elasticsearch-rest-high-level-client</artifactId>
  <version>7.9.0</version>
</dependency>
```

Cliente ElasticSearch

A continuación se muestra como crear un cliente de Elasticsearch:

```
RestHighLevelClient client=new RestHighLevelClient(
RestClient.builder(new HttpHost("localhost", 9200,"http")));
```

Para liberar recursos invocaremos el método close(), cuando lo dejemos de utilizar:

```
client.close();
```

Cliente ElasticSearch BasicAuth

Si se tiene autenticación, crearemos el cliente del siguiente modo:

Define las credenciales a utilizar:

```
final CredentialsProvider credentialsProvider=new
BasicCredentialsProvider();
credentialsProvider.setCredentials(AuthScope.ANY,new
UsernamePasswordCredentials("user","password"));
```

Asigna las credenciales utilizando el RestClient.builder:

```
RestHighLevelClient client=new
RestHighLevelClient(RestClient.builder(new
HttpHost("localhost", 9200,"http"))
.setHttpClientConfigCallback(new
HttpClientConfigCallback() {
  @Override
  public HttpAsyncClientBuilder
      customizeHttpClient(HttpAsyncClientBuilder
      httpClientBuilder) {
      return httpClientBuilder.setDefaultCredentialsProvider
            (credentialsProvider);
  }
}));
```

Creando documento

Una vez creado el cliente el siguiente paso es indexar algunos documentos, para esto utilizaremos el siguiente código:

Crear documento para indexar.

```
IndexRequest request= new
IndexRequest("devs4j-transactions");

request.id("3");
request.source("{\"nombre\":\"Sammie\","
      +"\"apellido\":\"Goldner\","
      +"\"username\":\"hugh.vonrueden\","
      +"\"monto\":9622235.2009}", XContentType.JSON);
```

Indexando de forma síncrona

Si se desea indexar el documento de forma síncrona es posible utilizar el siguiente método:

```
client.index(request, RequestOptions.DEFAULT);
```

DEVS4J
Hecho por Devs para Devs

Indexando de forma asíncrona

Si se desea indexar el documento de forma asíncrona es posible utilizar el siguiente método:

```
client.indexAsync(request, RequestOptions.DEFAULT,new
ActionListener<IndexResponse>() {
 @Override
public void onResponse(IndexResponseindexResponse) {
  //Successful response
 }
 @Override
 public void onFailure(Exception e) {
  //Failure response
 }
});
```

Realizando búsquedas

Si se desea realizar una búsqueda es posible utilizar el siguiente endpoint:

GET /devs4j-transactions/_search

```
{
  "query": {
      "match": {
          "nombre": "Johnson"
      }
  }
}
```

La petición anterior devolverá todos los registros con nombre Johnson.

Ordenando resultados

Si se desea realizar una búsqueda y ordenar la respuesta es posible utilizar el siguiente endpoint:

GET /devs4j-transactions/_search

```
{
  "query": {
      "match": { "nombre": "Usha" }
    },
  "sort": [
    { "monto": "desc" }
  ]
}
```

La petición anterior devolverá todos los registros con nombre Usha ordenados por monto.

Bulk Index

Indexa múltiples documentos al mismo tiempo con :

```
BulkRequest request=new BulkRequest();

IndexRequest request= new
IndexRequest("devs4j-transactions");

request.add(indexRequest);

client.bulkAsync(request,
RequestOptions.DEFAULT,listener);
```