**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

Descarga kafka

Para descargar apache kafka debes acceder a la siguiente url :

[bttps'//katka apache org/dowoloads](https://kafka.apache.org/downloads)

Asegurate de descargar la versión que dice binary downloads.

Inicia kafka

Crear un producer

Para iniciar un producer ejecutaremos el siguiente comando:

$ bin/ kafka-console-producer .sh - - topi c devs4j - topic - -bootstrap-ser ver

local host :9092

Crear un consumer

Para iniciar un consumer ejecutaremos el siguiente comando:

$ bin/ kafka -console-consumer .sh - - topi c devs4j - topic - - f rom-begi nni ng

* -bootstrap- server ocalhos t :9092
* -proper ty print .key=true - -proper ty key . separator=" - "

El parámetro - - f rom- beginning permite especificar si queremos

recibir solo los mensajes nuevos o queremos leer todos desde el inicio.

Topics / Partitions / Offsets

Replicas

Creando un topic

Untopic 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).

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

Untopic 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).

$ bin/ kaf ka - topi cs .sh

* -boots t rap- se rver

local host :9092 - -c reate - - topic devs4 j - topic - -partitions 5

* - replication - f actor 1

Este comando recibe los siguientes parámetros:

* bootstrap- serve r = Kafkaserver
* topie = Nombre del topic a crear
* part í tions = Número de particiones
* replica tion- factor = Número de réplicas

Modificando un topic

Para modificar un topic existente ejecutaremos el siguiente comando:

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

$ bin/ zookeeper - server - start .sh conf ig/ zookeeper .proper ties

$ bin/ kaf ka - server- s tar t .sh conf ig/ server .proper t ies

Esto iniciará tanto zookeeper como kafka.

$ bin/ kaf ka- topi c s.sh - -bootstrap- server

local host :9092 - -alter - - topi c devs4j - topic \

- -par titions 40

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

Agregando configuraciones

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

$ bin/ kafka-conf igs .sh - -bootstrap- server local host :9092 - - enti ty- type topi cs

* -entity -name devs4j - topi c - -alter
* -add-conf ig x=y

por broker

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 (ln-sync) y se mantiene actualizada con los cambios del líder. Si el líder falla una ln-sync se puede convertir en lider.

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

Listando topics

Quitando configuraciones

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

Puedes listar los topics disponibles ejecutando:

$ bin/ kaf ka - topi cs .sh - -list

- -bootstrap- server

local host :9092

Salida de ejemplo: dev s4 j - topi c

$ bin/ kaf ka-conf igs .sh - - bootstrap- server local host :9092 - -enti ty - type topi cs

* -entity -name devs4j - topi c - -alter
* -delete-config x

Borrando un topic

Para borrar un topic existente ejecutaremos el siguiente comando:

Producer

Consumer

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

* Control + C en los consumers y producers
* Control + cen los brokers de kafka
* Control + C en elservidor de zookeeper Si se desea borrar la información,ejecutar:

$ rm -rf / tmp/kafka- ogs / tmp/ zookeeper

Ver definición de un topic

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

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

$ bin/ kaf ka - topi c s .sh - -descr ibe

* - topi c devs4j - topic
* -bootstrap- server

$ bin/kaf ka- topi cs .sh - -bootst rap- server

localhos t :9092 - - delete - - topi c devs 4j - topi c Kafka no permite reducir el número de particiones.

local host :9092

Un consumer lee mensajes de uno o más topics y los procesa. La diferencia entre el la posición actualdel 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).

Salida de ejemplo:

Topie: devs4j-topie PartitionCount: 5 ReplicationFaetor: 1

Consumer groups

Topie: devs4j-topie Partition: O Leader:O Replicas:O lsr: O

Tapie: devs4j-tapie Partitian:1 Leader:o Replicas:o lsr: O

Tapie: devs4j-topie Partition:2 Leader:O Replicas:o lsr:o

Tapie: devs4j-tapie Partitian:3 Leader:O Replicas:o lsr:o

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.

***O*** www .twitter.com/devs4j

Tapie: devs4j-topie Partition:4 Leader:O Replicas: O lsr:O

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.



# DEVS J

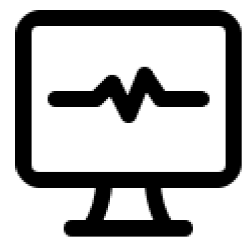
##### www .facebook.com/devs4j www .devs4j.com

**Apache Kafka - Diagramas**

**Kafka use cases Replicas**

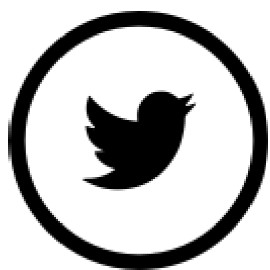
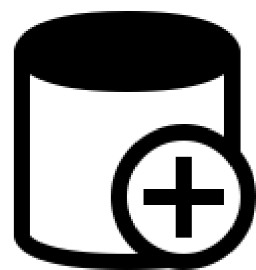
o **o**

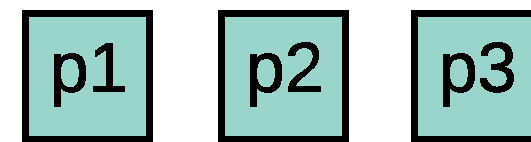
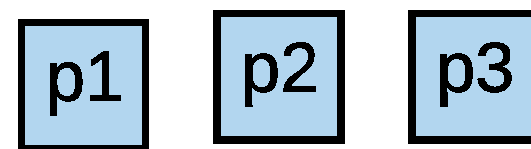
Repli catian factor =2



Broker 1 1 1 Broker 2 1 1 Broker 3

Apache Kafka

  o **láV**  Tapie 1 Tapie **1**



I Leader I

**t**

IFollower !

)

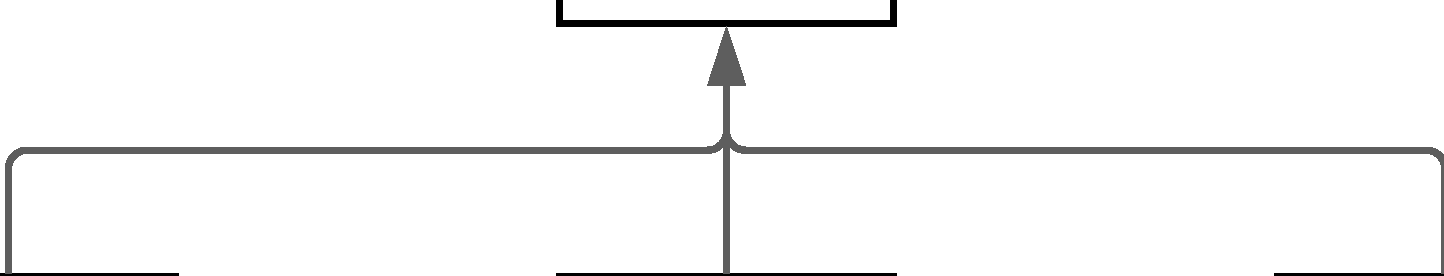
**Kafka Cluster Estructura de un mensaje**

Key = 10101

Value = Message

Timestamp = 10/08/2020TOO:OO:OO

Cluster



Broker 1 Broker 2 Broker 3

#### Flujo de entrega de mensajes

Consumer group 1

|  |  |  |
| --- | --- | --- |
|  | **Topie**    1 Partitian 1 1 10111213141515171  1 Partitian 2 @E§ | |
| I Producer I  + |
| IMessage I  Key = 10101  Value = Message I |
| 1  1 | 1 |
|  |
| Partitian n 1 **1°11121314151** |
|  |

Consumer

-

1 **1**

Consumer

1 **1**

•

1 **1**

Consumer

1 **1**

Consumer

Consumer group 2

-



***O*** [**www.twitter.com/devs4j**](http://www.twitter.com/devs4j)[**www.facebook.com/devs**](http://www.facebook.com/devs) **4j**

**DEVS J**

[**www.devs4j.com**](http://www.devs4j.com/)

Apache Kafka - Procesamiento de mensajes

configuración

Java COnsumer

Produce, transaccional

El código para consumir mensajes desde Java es el siguiente:

Properties props=new PropertiesQ; props.setProperty("bootstrap.servers","localhost:9092''); props.setProperty(''group.id","devs4j-group''); props.setProperty(''enable.auto.commit'',''lrue"): props.setProperty(''auto.commit.interval. ms","1000''); props.setProperty(''key.deserializer", "org.apache.l<alka.common.serialization.StringDeselializer''): props.setProperty(''value.deserializer", "org.apache.l<alka.common.serialization.StringDeserializer'');

tyy(KalkaConsumer<String, String> consumer=new KalkaConsumer<>(props):) {

consumer.subscribe(Arrays.asLis!("devs4j-topic'')); whi e(tTue){

ConsumerRecords<String,String>records= consumer .poll(Duration.ofMíllis(lOO));

for(ConsumerRecord<String, String>record

:records)

log.infoC'offset =O, key ={}, value =O",

record.offsetQ,record.keyQ,record.valueO);

Consumer properties

Para enviar mensajes es necesalio agregar las siguientes dependencias:

<dependencias >

<dependency>

<groupld>org.apache.kafka</groupld>

<artifactld>kafka-<:lients<lartifactld>

<version>2.3.0<lversion>

<ldependency>

<dependency>

<groupld>org.slf4j<lgroupld>

<artifactld>slf4j-simple</artifactld>

<version>l.6.1<lversion>

<ldependency>

</dependencias>

De igual modo incluiremos el siguiente pluging pera especificar laversión de Java.

<build>

<plugins>

<plugin>

<groupld>org.apache. maven.plugins</groupld>

<artifactld>maven-compiler-plugin<lartifactld>

<configuralion>

<source> .8</source>

<target> .B<ltarget>

</configuration>

</plugin>

</plugins>

<lbuild>

A continuación las modificaciones para convertir el producer en transaccional:

Properties props=new PropertiesQ; props.putC'bootstrap.servers","localhost:9092"); props.putC'acks","all"); props.putC'compression.type","gzip''): props.putC'linger.ms","1");

props.putC'batch.size","32384''); props.putC'lransactional .id"."devs4j-producer' '); props.putC'buffer.memory",''33554432''); props.putC'key.serializer'',

"org.apache.kall<a.common.serializalion.StringSerializer' '): props.putC'value.serializer", "org.apache.kalka.common.serialization.StringSerializer' ');

try{Producer<String, String> producer = new KalkaProducer<>(props)) {

try{ producer.initTransactionsQ; producer.beginTransactionQ;

for(inti= O;i< OOOOO; i++) { producer.send(new

ProducerRecord<> C'devs4j-topic'' ,"message")): ifO== 100) {

throw new ExceptionC'Random exception'');

}

producer.commitTransactionQ;

]catch(Exceplion e) { producer.abortTransactionQ;

log.error("Error processing messages" ,e);

}

}

Java Produce,

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 (mico que determina el consumer group al que pertenece el consume,.
* enable.auto.commit - Si es verdadero se realizaré commit alos offsets de forma periodica en segundo plano.
* auto.commit.interval.ms: Lafrecuencia en milisegundos en que se realizaré commit a los offsets (enable.auto.commit debe ser true)
* key.deserializer - Define la clase Java que va a deserializar la llave.

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

Consumer transaccional

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.StlingSerializer''); props.put(''value.serializer" , "org.apache.kafka.common.serialization.StlingSerializer'');

try(Producer<Stling, String>producer=new KafkaProducer<>(props);) {

for(inti= O; i< 100 ;i++) { producer.send(new

ProducerRecord<> ("devs4j-topic","key'',"message"));

}

producer.flushQ;

A continuación las modificaciones pera convertir el consumer entransaccional:

Properties props=new PropertiesQ; props.setProperty("bootstrap.servers","localhost9092'1: 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.kalka.common.serialization.StringDeserializer''); props.setProperty("value.deserializer", "org.apache.kall<a.common.serializalion.StringDeserializer'');

props.setProperty("isolation.level","read\_committed'');

try(KafkaConsumer<String, String>eonsumer=new KalkaConsumer<>(props);) {

consumer.subscribe(Arrays.asList(''devs4j-topic'')): while(true) {

ConsumerRecords<String,String>records= consumer .poll(Dura!ion.ofMí//ís(lOO)): for(ConsumerRecord<String,

String>record:records)

log.infof'parlilion = O , offset = O, key = O, value = O",record.partitionQ,record.offsetQ, record.keyQ,record.valueQ):

Produce, properties

Transactional



**DEVS J**

[www.facebook.com/devs4j](http://www.facebook.com/devs4j)

A continuación se listan algunas propiedades importantes a configurar enlos producers:

* bootstrap.servers - Server + Puerto de todos los brokers del cluster
* acks - Definesi se requiere un ack (confirmación de recibido) del mensaje, los valores posibles son O (no), 1 (si, de un broker), ali(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 tamano méximo del buffer en el que se pueden colocat batches,

***O*** [www.twitter.com/devs4j](http://www.twitter.com/devs4j)

Es posible manejar transacciones utilizando kafka, para esto

se deben seguir los siguientes puntos:

* Producer

-Definir acks =all

-Asignar untransactional 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 pasarén

de read\_uncommitted a read\_committed y serán leidos por el consumer.

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

* producer.initTransactions{);
* producer.beginTransaclionQ;
* producer.commitTransactionQ;
* producer.abortTransactionQ;

www .devs4j.com

Callbacks

**Apache Kafka - Conceptos avanzados**

El problema del ordenamiento

Ejecutando consumer threads

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](https://kafka.apache.org/documentation/#retries) > e} al enviar un mensaje, existe la posibilidad de que se vaya a la partición correcta pero se pierda el orden.

Debido alo anterior, si es totalmente necesario que se preserve el orden de los mensajes, es recomendable que se configure max.in.fl ght.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 = 1el performance o throughput del producer se verá impactado.

Para ejecutar los threads definidos anteriormente ejecutaremos el siguiente código:

public class Devs4jMultithreadConsumer { public static void main(StringOargs) {

//Config properties ExecutorService executor= *Executors.newFíxedThreadPoo/(5);* for(inti= O;i< 5;i++) {

Runnable worker= new

Devs4jThreadConsumer( newKafkaConsumer<>(props)) executor.execute(worker);

}

executor.shutdownO; while(!executor .isTerminatedO) ;

}

Multi thread consumer

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

CallbackO {

@Override

public void onCompletion(RecordMetadata metadata, Exceptionexception) {

if(exceptionl=null) {

/og.error("Error sending the message"

,exception);

}

/og.info("Partition = {} Offset ={}", metadata.partitionQ,metadata.offsetO);

}

});

Utilizando lambdas: producer.send(new

ProducerRecord<>("devs4j-topic","message"),

(metadata,exception) -> { if(exception !=null) {

/og.error("Error sending the message ", exception);

}

/og.info("Partition = {} Offset ={}", metadata.partitionQ,metadata.offsetO) ;

});

Assign

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

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

prívate final AtomicBoolean closed = new AtomicBoolean( false);

public static final Logger /og= LoggerFactory.getLogger(Devs4jThreadConsumer

.class);

public Devs4jThreadConsumer (KafkaConsumer<String, String>consumer) {

this.consumer=consumer ;

@Override

public void runo { try{

consumer.subscribe(Arrays.asList(''devs4j-topic")); while(!closed.getO) {

ConsumerRecords<String, String>records=

consumer.poll(Duration.ofMi//i (lOOOO)); for(ConsumerRecord<String, String>record

:records) {

/og.info("offset = {}, key = {}, value =

{}",record.offsetO,record.keyO,record.valueO);

}

}

}catch(WakeupException e) { if(!closed.getO)

throw e;

}finally{

consumer.closeQ;

}

}

public void shutdownO { closed.set(true); consumer.wakeupQ;

El método assign([Collection](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html?is-external=true)<T[opicPartition](https://kafka.apache.org/0100/javadoc/org/apache/kafka/common/TopicPartition.html)> 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", O);

consumer.assign(Arrays.asList(topicPartition));

Seek

Asegurar el orden

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",O);

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

while(true) {

ConsumerRecords<Stri ng, String>records= consumer .poll(Duration.ofM//is(lOO)); for(ConsumerRecord<Stri ng,

String>record:records)

/og.info("partition = {} , offset = {}, key = {}, value = {}",record.partitionQ,record.offsetO, record.keyQ,record.valueO) ;

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

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

Disable auto commit

Enviar a un tapie con 5 particiones mensajes con la

siguiente estructura:

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.

ldempotent producers

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 commitSyncO;.

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.

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.request s.per.connection < = 5
* retries > O
* acks > ali

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

***O*** www .twitter.com/devs4j

www .facebook.com/devs4j

**DEVS J**

www .devs4j.com

Configuración

Para configurar Spring kafka se debe incluir la siguiente dependencia:

<dependency>

<groupld>org.springframework. kafka</groupld>

<artifactld>spring-kafka</artifactld>

</dependency>

Apache Kafka - Integración con Spring

Producer properties

Envío de mensajes síncrono

Consumer properties

A continuación se muestra como enviar mensajes de forma síncrona:

kafkaTemplate.send(new ProducerRecord<String,Stri ng> ("key","value")).getQ;

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

prívate Map<String, Object> producerPropsQ { Map<String. Object> props=new HashMap<>Q;

props.put(ProducerConf ig.BOOTSTRAP\_SERVERS\_CONF/G,

"localhost:9092"); props.put(ProducerConfig.RETR/ES\_ CONF/G,O);

props.put(ProducerConfig .BATCH\_S/ZE\_CONF/G, 16384); props.put(ProducerConfig.LJNGER\_MS\_CONFIG,l); props.put(ProducerConf ig.BUFFER\_MEMORY\_CONF/G,33554432); props.put(Producerconfig.KEY\_SER/ALIZER\_CLASS\_CONF/G, lntegerSerializer.class):

*props .put(ProducerConl ig.VALUE\_SERIALIZER\_CLASS\_CONFIG,*

StringSerializer.class);

return props;

@Bean

publlc KafkaTemplate<lnleger. String> createTemplateQ { Map<String, Object>senderProps= producerPropsQ; ProducerFactory<lnteger, String> pi= new DefaultKafkaProducerFactory<l nteger, String>(senderProps);

KafkaTemplate<lnteger, String> template=new KafkaTemplate<>( pf); return template ;

Se incluyen las siguientes nuevas propiedades:

* *RETRIES\_CDNFIG:Detine* los reintentos que se realizarán en caso de error.
* *BATCH\_SIZE\_ CONFIG:EI* 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.

kafkaTemplate.send(new ProducerRecord<String,String> ("key'',''value")).get(lO,TimeUnit.SECONOS);

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

Utilizando el producer

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",groupld

="devs4j-group3" , properties =

{"max.poli.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){

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

@Bean

public Map<String, Object> consumerPropsQ { Map<String, Object>props=new HashMap<>O;

props.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONF/G, "localhost:9092"):

props.put(ConsumerConfig. GROUP\_JD\_CONF/G, "group'1:

props.put(ConsumerConfig.ENABLE\_AUTO\_COMMIT\_CONFIG,

true):

props.put(ConsumerConfig.AUTO\_ COMMIT\_ /NTERVAL\_MS\_ CONF/G, "100'1;

props.put(ConsumerConfig.SESS/ON\_ T/MEOUT\_MS\_ CONF/G,

"15000'1;

*props .put(Consumerconlig.KEY\_DESERIALJZER\_CLASS\_CONFIG ,*

lntegerDeserializer. class);

*props .put(ConsumerConlig. VALUE\_DESERIALIZER\_CLASS\_CONFIG ,*

StringDeserializer.class): return props;

A continuación laexplicación de cada una:

*-BOOTSTRAP\_SERVERS\_CONFIG :* Lista de brokers de kafka en el cluster.

- *GROUP\_tD\_CONFIG,* Consumer group que consumirá los mensajes

*-ENABLE\_Auro\_coMMIT\_CONRG :* Determina si se hará commit al offset de forma periódica

*-Auro\_COMMff\_lNTERVAL\_Ms\_cONFIG :* Determina lafrecuencia en milisegundos en la que se hará commit a los offsets, solo es necesaria si

*ENABLE\_AUTO\_ COMMIT\_CONFJG =true.*

*-sESstON\_TIMEDUT\_Ms\_cDNFtG ,*Timeout utilizado para determinar errores en los clientes.

*-KEY\_DESERIALIZER\_CLASS\_CONFIG:* Clase a utilizar para deserializar la llave

*-VALUE\_DESERIALJZER\_CLASS\_CONFIG:* Clase a utilizar para deserializar el mensaje

Lectura de mensajes de kafka

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

@Autowired

prívate KafkaTemplate<lnteger. String> kafkaTemplate;

public Role createRole(Rolerole) { Role save = repository.save(role);

kafkaTemplate.send("devs4j-topic",save.getNameQ) ; 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

Unavez definidas las propiedades se debe configurar el listener:

@Bean

public ConsumerFactory<lnteger, String> consumerFactoryQ { return new

DefaultKafkaConsumerFactory<>( consumerPropsQ);

}

@Bean

public ConcurrentKafkalistenerContainerFactory<lnteger, String> kafkalistenerContainerFactoryQ {

ConcurrentKafkaListenerContainerFactory<l nteger. String> factory = new ConcurrentKafkalistenerContainerFactory<>Q ;

factory.setConsumerFactory(consumerFactoryQ); return factory;

}

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

@Kafkalistener(topics ="devs4j-topic", groupld ="consumer') public void listen(String message) {

System.out.println("Received Messasge in group foo:

"+message);

}



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>O {

@Override

public void onSuccess(SendResult<String.String>result) {

/og.info("Message sent");

@Override

public void onFailure(Throwable ex) { fog.error('Error sending message ",ex);

@Override

public void onFailure(KafkaProducerExcepti on ex) { fog.error('Error sending message ",ex);

});

***O*** www .twitter.com/devs4j

www .facebook.com/devs4j

www .devs4j.com

**DEVS J**

Apache Kafka - Integración con Spring Pt2

Consumiendo mensajes completos

Creaci6n de meter registry

Mostrando métricas

@Scheduled(fixedDelay = 2000, initialDelay = 500) public void messageCountMetricQ {

double count =

meterRegistry.get("kalka.producer.record.send.total'1

.functionCounterQ.countQ; tog.infof'Count O",count);

lntegraci6n con producer

Enel 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 lainformación:

publ c void listen(List<ConsumerRecord<String, String»messages){ for(ConsumerRecord<String, String>message :

messages){

tog.infof'Offset O Partition= O Key = OValue = O", message.offsetQ,message. message.keyQ,message.valueO);

Unavez configurada la dependenciade micrometer *y* prometheus, crearemos un bean de tipo MetrlcReglstry como se muestra acontinuación:

@Bean

public MeterRegistry meterRegistryQ {

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

}

Métt'icas disponibles

Multithread

Paraconocer las métricas disponibles podremos ejecutar el método MelricRegistry.getMetersQ como se muestra a continuación:

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

Algunas métricas disponibles:

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

Elsiguiente paso será integrar el MeterRegistry en nuestro producer como se muestra acontinuación:

@Bean

publlc KafkaTemplate<String, String> kalkaTemplateQ { DefaultKalkaProducerFactory<String, String> producerFactory= new DefaultKafkaProducerFactory<>(

producerPropertiesO); producerFactory .addlistener(new

MicrometerProducerlistener<String,String> (meterRegistryQ));

KafkaTemplate<String, String> template = new KelkaTemplate<>(producerFactory);

returntemplate;

}

Lo importante amencionar en este punto es lainclusión del método meterRegistryQ en la creación de un MicrometerProducerlistener, quien estará dando seguimiento alas métricas.

Si se desea consumir mensajes de forma concurrente en Spring se debe agregar la siguiente configuración en el COncum1ntK81kal.istenerContainerFactory:

listenerContainerFactory .setConcurrency(3);

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

Los nombres delos hilos se consttuirán por default del siguiente modo:

* container-O-C-2 - Primer hilo
* container-1-C-2 - Segundo hilo
* container-2-C-2 - Tercer hilo

Habilitando consumer en tiempo de eiecuci6n

Habilitando scheduling

En esta sección utilizaremos la anotación @Scheduled así que es necesario que en nuestra clase principal o enla de configuración coloquemos la anotación

@EnableScheduling, como se muestra a continuación:

@SpringBootApplication

@EnableScheduling

publlc class CUrsoKalkaSpringApplication {

*11* •..

Inyectando el MetricRegistry

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

@Kalka.Listener(id ="autoStartup", autoStartup ="false')

- id: Define unidentificador para nuestro container

- autoStartup: Define que nuestro consumer noiniciará por defecto cuando inicie nuestra aplicación.

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

@Autowired

pñvate KalkaListenerEndpointRegistry registry;

publlc vold anyMethodQ {

log.infoC'Starting consuming messages'J; registty.getListenerContainerC'autoStartup").startQ;

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

@Autowired

prlvate MeterRegistty meterRegistry;

Uso de métricas

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

<dependency>

<groupld>io.micrometer<Jgroupld>

<artifactld>micrometer-registry-prometheus <Jartifactld:

<Jdependency>

Ladependencia anterior nos proporcionará el soporte para trabajar con métricas utilizando micrometer *y* prometheus.

Producer

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

@Scheduled(fixedDelay = 2000, initialDelay = 100) publlc vold scheduleQ {

log.infoC'Sending messages "); for(inti= O;i< 200;i++) {

kalkaTemplate.send('devs4j-topic" ,"Message"+i);

Precauci6n

Si no hay métricas disponibles se producirá una MeterNotFoundExceptlon, as! que es importante contemplar ese caso durante el desarrollo de nuestras aplicaciones.

***O*** [www.twitter.com/devs4j](http://www.twitter.com/devs4j) [www.facebook.com/devs4j](http://www.facebook.com/devs4j)



**DEVS J**

www .devs4j.com

**Apache Kafka - ElasticSearch**

Iniciando Elasticsearch

Cliente ElasticSearch

Indexando de forma asíncrona

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 closeQ, cuando lo dejemos de utilizar:

client.closeQ;

Cliente ElasticSearch BasicAuth

Descarga elasticsearch:

curl -L -O https ://artifacts .elastic .co/ downloads/elasticsearch

/elasticsearch-7.9.0-darwi n-x86\_64.tar .gz

Descomprime el archivo:

tar - xvf

elasticsearch-7.9.0-darwi n- x86\_64. tar .gz

Inicia Elasticsearch:

cd elasticsearch-7 .9 .0/bin

./elasticsearch

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

client.indexAsync(request, RequestOptions .DEFAULT,new ActionListener<lndexResponse>Q {

@Override

public void onResponse(lndexResponse indexResponse) {

//Successful response

}

@Override

public void onFailure(Exception e) {

//Failure response

}

});

Validando salud

Realizando búsquedas

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

Define las credenciales a utilizar :

final CredentialsProvider credentialsProvider =new BasicCredentialsProviderQ;

credentialsProvider .setCredentials(AuthScope .ANY,new UsernamePasswordCredentials(" user","password"));

Asigna las credenciales utilizando el RestClient.builder :

RestHighLevelClient client=new RestHighLevelClient(RestClient.bui /der(new HttpHost("localhost", 9200,"http''))

.setHttpClientConfigCallback( new HttpClientConfigCallbackQ {

@Override

public HttpAsyncClientBuilder customizeHttpClient(HttpAsyncClientBuilder httpClientBuilder) {

return httpClientBuilder .setDefaultCredentialsProvider (credentialsProvider );

}

}));

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

GET /\_cat/health?v

Indexando documentos

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

GET /devs4j-transacti ons/\_search

{

"query": {

"match": {

"nombre": "Johnson"

}

}

La petición anterior devolverá todos los registros con nombre Johnson.

Ordenando resultados

Para indexar documentos ejecutaremos el siguiente endpoint:

PUT /devs4j-transacti ons/transactions/1

{

"nombre":"Noah",

"apellido":"Ku/as", *"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 .

Creando documento

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

GET /devs4j-transacti ons/\_search

{

"query": {

"match": { "nombre": "Usha" }

},

"sort": [

{ "monto": "dese" }

l

}

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

Consultando documento por id

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

Crear documento para indexar.

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

request.id("3"); request.source("{\"nombrel": "Sammie\","

+''l"apellidol":l"Goldnerl","

+''l"usernamel": "hugh.vonruedenl","

+''l"montol":9622235 .2009}", XContentType .JSONJ;

Indexando de forma síncrona

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

GET /devs4j-transacti ons/transactions/1

Esto nos devolverá una salida como la siguiente :

*"\_index":"devs4j-transactions",* "\_type":*"transactions",* "\_id":"1",

"\_version":1,

"\_seq\_no" :O, "\_primary\_term":1, "found":true,

"\_source": {

"nombre":*"Noah",*

"apellido":"Ku/as", *"username" :"tristan./owe" ,* "monto":1.9829264751E7

Bulk lndex

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

client.index(request, Request0ptions .DEFAUL7);

Configurando proyecto



Indexa múltiples documentos al mismo tiempo con :

BulkRequest request=new BulkRequestQ;

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

request.add(indexRequest);

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

Agregaremos la siguiente dependencia:

<dependency>

<groupld>org.elasticsearch .client</groupld>

<artifactld>elasticsearch-rest-high-level-client </artifactld>

<version>7.9.0</version>

</dependency>

***O*** [www.twitter.com/devs4j](http://www.twitter.com/devs4j)

**DEVS J**

www.facebook.com/devs4j

[www.devs4j.com](http://www.devs4j.com/)