package com.twitter.unified\_user\_actions.service.module

import com.google.inject.Provides

import com.twitter.decider.Decider

import com.twitter.decider.SimpleRecipient

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finatra.kafka.producers.BlockingFinagleKafkaProducer

import com.twitter.finatra.kafka.serde.ScalaSerdes

import com.twitter.finatra.kafka.serde.UnKeyed

import com.twitter.finatra.kafka.serde.UnKeyedSerde

import com.twitter.iesource.thriftscala.InteractionEvent

import com.twitter.inject.TwitterModule

import com.twitter.inject.annotations.Flag

import com.twitter.kafka.client.headers.Zone

import com.twitter.kafka.client.processor.AtLeastOnceProcessor

import com.twitter.unified\_user\_actions.adapter.AbstractAdapter

import com.twitter.unified\_user\_actions.adapter.uua\_aggregates.RekeyUuaFromInteractionEventsAdapter

import com.twitter.unified\_user\_actions.kafka.ClientConfigs

import com.twitter.unified\_user\_actions.kafka.ClientProviders

import com.twitter.unified\_user\_actions.kafka.CompressionTypeFlag

import com.twitter.unified\_user\_actions.thriftscala.KeyedUuaTweet

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.StorageUnit

import com.twitter.util.logging.Logging

import org.apache.kafka.clients.consumer.ConsumerRecord

import org.apache.kafka.clients.producer.ProducerRecord

import org.apache.kafka.common.header.Headers

import org.apache.kafka.common.record.CompressionType

import javax.inject.Singleton

import javax.inject.Inject

object KafkaProcessorRekeyUuaIesourceModule extends TwitterModule with Logging {

override def modules = Seq(FlagsModule)

private val adapter = new RekeyUuaFromInteractionEventsAdapter

// NOTE: This is a shared processor name in order to simplify monviz stat computation.

private final val processorName = "uuaProcessor"

@Provides

@Singleton

@Inject

def providesKafkaProcessor(

decider: Decider,

@Flag(FlagsModule.cluster) cluster: String,

@Flag(FlagsModule.kafkaSourceCluster) kafkaSourceCluster: String,

@Flag(FlagsModule.kafkaDestCluster) kafkaDestCluster: String,

@Flag(FlagsModule.kafkaSourceTopic) kafkaSourceTopic: String,

@Flag(FlagsModule.kafkaSinkTopics) kafkaSinkTopics: Seq[String],

@Flag(FlagsModule.kafkaGroupId) kafkaGroupId: String,

@Flag(FlagsModule.kafkaProducerClientId) kafkaProducerClientId: String,

@Flag(FlagsModule.kafkaMaxPendingRequests) kafkaMaxPendingRequests: Int,

@Flag(FlagsModule.kafkaWorkerThreads) kafkaWorkerThreads: Int,

@Flag(FlagsModule.commitInterval) commitInterval: Duration,

@Flag(FlagsModule.maxPollRecords) maxPollRecords: Int,

@Flag(FlagsModule.maxPollInterval) maxPollInterval: Duration,

@Flag(FlagsModule.sessionTimeout) sessionTimeout: Duration,

@Flag(FlagsModule.fetchMax) fetchMax: StorageUnit,

@Flag(FlagsModule.receiveBuffer) receiveBuffer: StorageUnit,

@Flag(FlagsModule.batchSize) batchSize: StorageUnit,

@Flag(FlagsModule.linger) linger: Duration,

@Flag(FlagsModule.bufferMem) bufferMem: StorageUnit,

@Flag(FlagsModule.compressionType) compressionTypeFlag: CompressionTypeFlag,

@Flag(FlagsModule.retries) retries: Int,

@Flag(FlagsModule.retryBackoff) retryBackoff: Duration,

@Flag(FlagsModule.requestTimeout) requestTimeout: Duration,

@Flag(FlagsModule.enableTrustStore) enableTrustStore: Boolean,

@Flag(FlagsModule.trustStoreLocation) trustStoreLocation: String,

statsReceiver: StatsReceiver,

): AtLeastOnceProcessor[UnKeyed, InteractionEvent] = {

provideAtLeastOnceProcessor(

name = processorName,

kafkaSourceCluster = kafkaSourceCluster,

kafkaGroupId = kafkaGroupId,

kafkaSourceTopic = kafkaSourceTopic,

commitInterval = commitInterval,

maxPollRecords = maxPollRecords,

maxPollInterval = maxPollInterval,

sessionTimeout = sessionTimeout,

fetchMax = fetchMax,

receiveBuffer = receiveBuffer,

processorMaxPendingRequests = kafkaMaxPendingRequests,

processorWorkerThreads = kafkaWorkerThreads,

adapter = adapter,

kafkaSinkTopics = kafkaSinkTopics,

kafkaDestCluster = kafkaDestCluster,

kafkaProducerClientId = kafkaProducerClientId,

batchSize = batchSize,

linger = linger,

bufferMem = bufferMem,

compressionType = compressionTypeFlag.compressionType,

retries = retries,

retryBackoff = retryBackoff,

requestTimeout = requestTimeout,

statsReceiver = statsReceiver,

trustStoreLocationOpt = if (enableTrustStore) Some(trustStoreLocation) else None,

decider = decider,

zone = ZoneFiltering.zoneMapping(cluster),

maybeProcess = ZoneFiltering.noFiltering

)

}

def producer(

producer: BlockingFinagleKafkaProducer[Long, KeyedUuaTweet],

k: Long,

v: KeyedUuaTweet,

sinkTopic: String,

headers: Headers,

statsReceiver: StatsReceiver,

decider: Decider,

): Future[Unit] =

if (decider.isAvailable(feature = s"RekeyUUAIesource${v.actionType}", Some(SimpleRecipient(k))))

// If we were to enable xDC replicator, then we can safely remove the Zone header since xDC

// replicator works in the following way:

// - If the message does not have a header, the replicator will assume it is local and

// set the header, copy the message

// - If the message has a header that is the local zone, the replicator will copy the message

// - If the message has a header for a different zone, the replicator will drop the message

producer

.send(new ProducerRecord[Long, KeyedUuaTweet](sinkTopic, null, k, v, headers))

.onSuccess { \_ => statsReceiver.counter("publishSuccess", sinkTopic).incr() }

.onFailure { e: Throwable =>

statsReceiver.counter("publishFailure", sinkTopic).incr()

error(s"Publish error to topic $sinkTopic: $e")

}.unit

else Future.Unit

def provideAtLeastOnceProcessor(

name: String,

kafkaSourceCluster: String,

kafkaGroupId: String,

kafkaSourceTopic: String,

commitInterval: Duration = ClientConfigs.kafkaCommitIntervalDefault,

maxPollRecords: Int = ClientConfigs.consumerMaxPollRecordsDefault,

maxPollInterval: Duration = ClientConfigs.consumerMaxPollIntervalDefault,

sessionTimeout: Duration = ClientConfigs.consumerSessionTimeoutDefault,

fetchMax: StorageUnit = ClientConfigs.consumerFetchMaxDefault,

fetchMin: StorageUnit = ClientConfigs.consumerFetchMinDefault,

receiveBuffer: StorageUnit = ClientConfigs.consumerReceiveBufferSizeDefault,

processorMaxPendingRequests: Int,

processorWorkerThreads: Int,

adapter: AbstractAdapter[InteractionEvent, Long, KeyedUuaTweet],

kafkaSinkTopics: Seq[String],

kafkaDestCluster: String,

kafkaProducerClientId: String,

batchSize: StorageUnit = ClientConfigs.producerBatchSizeDefault,

linger: Duration = ClientConfigs.producerLingerDefault,

bufferMem: StorageUnit = ClientConfigs.producerBufferMemDefault,

compressionType: CompressionType = ClientConfigs.compressionDefault.compressionType,

retries: Int = ClientConfigs.retriesDefault,

retryBackoff: Duration = ClientConfigs.retryBackoffDefault,

requestTimeout: Duration = ClientConfigs.producerRequestTimeoutDefault,

produceOpt: Option[

(BlockingFinagleKafkaProducer[Long, KeyedUuaTweet], Long, KeyedUuaTweet, String, Headers,

StatsReceiver, Decider) => Future[Unit]

] = Some(producer),

trustStoreLocationOpt: Option[String] = Some(ClientConfigs.trustStoreLocationDefault),

statsReceiver: StatsReceiver,

decider: Decider,

zone: Zone,

maybeProcess: (ConsumerRecord[UnKeyed, InteractionEvent], Zone) => Boolean,

): AtLeastOnceProcessor[UnKeyed, InteractionEvent] = {

lazy val singletonProducer = ClientProviders.mkProducer[Long, KeyedUuaTweet](

bootstrapServer = kafkaDestCluster,

clientId = kafkaProducerClientId,

keySerde = ScalaSerdes.Long.serializer,

valueSerde = ScalaSerdes.Thrift[KeyedUuaTweet].serializer,

idempotence = false,

batchSize = batchSize,

linger = linger,

bufferMem = bufferMem,

compressionType = compressionType,

retries = retries,

retryBackoff = retryBackoff,

requestTimeout = requestTimeout,

trustStoreLocationOpt = trustStoreLocationOpt,

)

KafkaProcessorProvider.mkAtLeastOnceProcessor[UnKeyed, InteractionEvent, Long, KeyedUuaTweet](

name = name,

kafkaSourceCluster = kafkaSourceCluster,

kafkaGroupId = kafkaGroupId,

kafkaSourceTopic = kafkaSourceTopic,

sourceKeyDeserializer = UnKeyedSerde.deserializer,

sourceValueDeserializer = ScalaSerdes.CompactThrift[InteractionEvent].deserializer,

commitInterval = commitInterval,

maxPollRecords = maxPollRecords,

maxPollInterval = maxPollInterval,

sessionTimeout = sessionTimeout,

fetchMax = fetchMax,

fetchMin = fetchMin,

receiveBuffer = receiveBuffer,

processorMaxPendingRequests = processorMaxPendingRequests,

processorWorkerThreads = processorWorkerThreads,

adapter = adapter,

kafkaProducersAndSinkTopics =

kafkaSinkTopics.map(sinkTopic => (singletonProducer, sinkTopic)),

produce = produceOpt.getOrElse(producer),

trustStoreLocationOpt = trustStoreLocationOpt,

statsReceiver = statsReceiver,

decider = decider,

zone = zone,

maybeProcess = maybeProcess,

)

}

}