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

import com.google.inject.Provides

import com.twitter.clientapp.thriftscala.LogEvent

import com.twitter.decider.Decider

import com.twitter.finagle.stats.StatsReceiver

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

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

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

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.client\_event.ClientEventAdapter

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

import com.twitter.unified\_user\_actions.kafka.serde.NullableScalaSerdes

import com.twitter.unified\_user\_actions.service.module.KafkaProcessorProvider.updateActionTypeCounters

import com.twitter.unified\_user\_actions.service.module.KafkaProcessorProvider.updateProcessingTimeStats

import com.twitter.unified\_user\_actions.service.module.KafkaProcessorProvider.updateProductSurfaceTypeCounters

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

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

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.StorageUnit

import com.twitter.util.logging.Logging

import javax.inject.Singleton

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

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

object KafkaProcessorClientEventModule extends TwitterModule with Logging {

override def modules: Seq[FlagsModule.type] = Seq(FlagsModule)

private val clientEventAdapter = new ClientEventAdapter

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

private final val processorName = "uuaProcessor"

@Provides

@Singleton

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.fetchMin) fetchMin: 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, LogEvent] = {

KafkaProcessorProvider.provideDefaultAtLeastOnceProcessor(

name = processorName,

kafkaSourceCluster = kafkaSourceCluster,

kafkaGroupId = kafkaGroupId,

kafkaSourceTopic = kafkaSourceTopic,

sourceKeyDeserializer = UnKeyedSerde.deserializer,

sourceValueDeserializer = NullableScalaSerdes

.Thrift[LogEvent](statsReceiver.counter("deserializerErrors")).deserializer,

commitInterval = commitInterval,

maxPollRecords = maxPollRecords,

maxPollInterval = maxPollInterval,

sessionTimeout = sessionTimeout,

fetchMax = fetchMax,

fetchMin = fetchMin,

processorMaxPendingRequests = kafkaMaxPendingRequests,

processorWorkerThreads = kafkaWorkerThreads,

adapter = clientEventAdapter,

kafkaSinkTopics = kafkaSinkTopics,

kafkaDestCluster = kafkaDestCluster,

kafkaProducerClientId = kafkaProducerClientId,

batchSize = batchSize,

linger = linger,

bufferMem = bufferMem,

compressionType = compressionTypeFlag.compressionType,

retries = retries,

retryBackoff = retryBackoff,

requestTimeout = requestTimeout,

statsReceiver = statsReceiver,

produceOpt = Some(clientEventProducer),

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

decider = decider,

zone = ZoneFiltering.zoneMapping(cluster),

)

}

/\*\*

\* ClientEvent producer is different from the defaultProducer.

\* While the defaultProducer publishes every event to all sink topics, ClientEventProducer (this producer) requires

\* exactly 2 sink topics: Topic with all events (impressions and engagements) and Topic with engagements only.

\* And the publishing is based the action type.

\*/

def clientEventProducer(

producer: BlockingFinagleKafkaProducer[UnKeyed, UnifiedUserAction],

k: UnKeyed,

v: UnifiedUserAction,

sinkTopic: String,

headers: Headers,

statsReceiver: StatsReceiver,

decider: Decider

): Future[Unit] =

if (ClientEventDeciderUtils.shouldPublish(decider = decider, uua = v, sinkTopic = sinkTopic)) {

updateActionTypeCounters(statsReceiver, v, sinkTopic)

updateProductSurfaceTypeCounters(statsReceiver, v, sinkTopic)

updateProcessingTimeStats(statsReceiver, v)

// 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[UnKeyed, UnifiedUserAction](

sinkTopic,

null,

k,

v,

headers.remove(Zone.Key)))

.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

}