package com.twitter.product\_mixer.component\_library.side\_effect

import com.twitter.conversions.DurationOps.\_

import com.twitter.conversions.StorageUnitOps.\_

import com.twitter.finatra.kafka.producers.FinagleKafkaProducerBuilder

import com.twitter.finatra.kafka.producers.KafkaProducerBase

import com.twitter.finatra.kafka.producers.TwitterKafkaProducerConfig

import com.twitter.product\_mixer.core.functional\_component.side\_effect.PipelineResultSideEffect

import com.twitter.product\_mixer.core.model.common.presentation.CandidateWithDetails

import com.twitter.product\_mixer.core.model.marshalling.HasMarshalling

import com.twitter.product\_mixer.core.pipeline.PipelineQuery

import com.twitter.stitch.Stitch

import com.twitter.util.Duration

import com.twitter.util.StorageUnit

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

import org.apache.kafka.common.serialization.Serializer

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

/\*\*

\* The Kafka publishing side effect.

\* This class creates a Kafka producer with provided and default parameters.

\* Note that callers may not provide arbitrary params as this class will do validity check on some

\* params, e.g. maxBlock, to make sure it is safe for online services.

\*

\* PLEASE NOTE: caller needs to add the following to the Aurora file to successfully enable the TLS

\* '-com.twitter.finatra.kafka.producers.principal={{role}}',

\*

\* @tparam K type of the key

\* @tparam V type of the value

\* @tparam Query pipeline query

\*/

trait KafkaPublishingSideEffect[K, V, Query <: PipelineQuery, ResponseType <: HasMarshalling]

extends PipelineResultSideEffect[Query, ResponseType] {

/\*\*

\* Kafka servers list. It is usually a WilyNs name at Twitter

\*/

val bootstrapServer: String

/\*\*

\* The serde of the key

\*/

val keySerde: Serializer[K]

/\*\*

\* The serde of the value

\*/

val valueSerde: Serializer[V]

/\*\*

\* An id string to pass to the server when making requests.

\* The purpose of this is to be able to track the source of requests beyond just ip/port by

\* allowing a logical application name to be included in server-side request logging.

\*/

val clientId: String

/\*\*

\* The configuration controls how long <code>KafkaProducer.send()</code> and

\* <code>KafkaProducer.partitionsFor()</code> will block.

\* These methods can be blocked either because the buffer is full or metadata unavailable.

\* Blocking in the user-supplied serializers or partitioner will not be counted against this timeout.

\*

\* Set 200ms by default to not blocking the thread too long which is critical to most ProMixer

\* powered services. Please note that there is a hard limit check of not greater than 1 second.

\*

\*/

val maxBlock: Duration = 200.milliseconds

/\*\*

\* Retries due to broker failures, etc., may write duplicates of the retried message in the

\* stream. Note that enabling idempotence requires

\* <code> MAX\_IN\_FLIGHT\_REQUESTS\_PER\_CONNECTION </code> to be less than or equal to 5,

\* <code> RETRIES\_CONFIG </code> to be greater than 0 and <code> ACKS\_CONFIG </code>

\* must be 'all'. If these values are not explicitly set by the user, suitable values will be

\* chosen. If incompatible values are set, a <code>ConfigException</code> will be thrown.

\*

\* false by default, setting to true may introduce issues to brokers since brokers will keep

\* tracking all requests which is resource expensive.

\*/

val idempotence: Boolean = false

/\*\*

\* The producer will attempt to batch records together into fewer requests whenever multiple

\* records are being sent to the same partition. This helps performance on both the client and

\* the server. This configuration controls the default batch size in bytes.

\* No attempt will be made to batch records larger than this size.

\* Requests sent to brokers will contain multiple batches, one for each partition with data

\* available to be sent. A small batch size will make batching less common and may reduce

\* throughput (a batch size of zero will disable batching entirely).

\* A very large batch size may use memory a bit more wastefully as we will always allocate a

\* buffer of the specified batch size in anticipation of additional records.

\*

\* Default 16KB which comes from Kafka's default

\*/

val batchSize: StorageUnit = 16.kilobytes

/\*\*

\* The producer groups together any records that arrive in between request transmissions into

\* a single batched request. "Normally this occurs only under load when records arrive faster

\* than they can be sent out. However in some circumstances the client may want to reduce the

\* number of requests even under moderate load. This setting accomplishes this by adding a

\* small amount of artificial delay&mdash;that is, rather than immediately sending out a record

\* the producer will wait for up to the given delay to allow other records to be sent so that

\* the sends can be batched together. This can be thought of as analogous to Nagle's algorithm

\* in TCP. This setting gives the upper bound on the delay for batching: once we get

\* BATCH\_SIZE\_CONFIG worth of records for a partition it will be sent immediately regardless

\* of this setting, however if we have fewer than this many bytes accumulated for this

\* partition we will 'linger' for the specified time waiting for more records to show up.

\* This setting defaults to 0 (i.e. no delay). Setting LINGER\_MS\_CONFIG=5, for example,

\* would have the effect of reducing the number of requests sent but would add up to 5ms of

\* latency to records sent in the absence of load.

\*

\* Default 0ms, which is Kafka's default. If the record size is much larger than the batchSize,

\* you may consider to enlarge both batchSize and linger to have better compression (only when

\* compression is enabled.)

\*/

val linger: Duration = 0.milliseconds

/\*\*

\* The total bytes of memory the producer can use to buffer records waiting to be sent to the

\* server. If records are sent faster than they can be delivered to the server the producer

\* will block for MAX\_BLOCK\_MS\_CONFIG after which it will throw an exception.

\* This setting should correspond roughly to the total memory the producer will use, but is not

\* a hard bound since not all memory the producer uses is used for buffering.

\* Some additional memory will be used for compression (if compression is enabled) as well as

\* for maintaining in-flight requests.

\*

\* Default 32MB which is Kafka's default. Please consider to enlarge this value if the EPS and

\* the per-record size is large (millions EPS with >1KB per-record size) in case the broker has

\* issues (which fills the buffer pretty quickly.)

\*/

val bufferMemorySize: StorageUnit = 32.megabytes

/\*\*

\* Producer compression type

\*

\* Default LZ4 which is a good tradeoff between compression and efficiency.

\* Please be careful of choosing ZSTD, which the compression rate is better it might introduce

\* huge burden to brokers once the topic is consumed, which needs decompression at the broker side.

\*/

val compressionType: CompressionType = CompressionType.LZ4

/\*\*

\* Setting a value greater than zero will cause the client to resend any request that fails

\* with a potentially transient error

\*

\* Default set to 3, to intentionally reduce the retries.

\*/

val retries: Int = 3

/\*\*

\* The amount of time to wait before attempting to retry a failed request to a given topic

\* partition. This avoids repeatedly sending requests in a tight loop under some failure

\* scenarios

\*/

val retryBackoff: Duration = 1.second

/\*\*

\* The configuration controls the maximum amount of time the client will wait

\* for the response of a request. If the response is not received before the timeout

\* elapses the client will resend the request if necessary or fail the request if

\* retries are exhausted.

\*

\* Default 5 seconds which is intentionally low but not too low.

\* Since Kafka's publishing is async this is in general safe (as long as the bufferMem is not full.)

\*/

val requestTimeout: Duration = 5.seconds

require(

maxBlock.inMilliseconds <= 1000,

"We intentionally set the maxBlock to be smaller than 1 second to not block the thread for too long!")

lazy val kafkaProducer: KafkaProducerBase[K, V] = {

val jaasConfig = TwitterKafkaProducerConfig().configMap

val builder = FinagleKafkaProducerBuilder[K, V]()

.keySerializer(keySerde)

.valueSerializer(valueSerde)

.dest(bootstrapServer, 1.second) // NOTE: this method blocks!

.clientId(clientId)

.maxBlock(maxBlock)

.batchSize(batchSize)

.linger(linger)

.bufferMemorySize(bufferMemorySize)

.maxRequestSize(4.megabytes)

.compressionType(compressionType)

.enableIdempotence(idempotence)

.maxInFlightRequestsPerConnection(5)

.retries(retries)

.retryBackoff(retryBackoff)

.requestTimeout(requestTimeout)

.withConfig("acks", "all")

.withConfig("delivery.timeout.ms", requestTimeout + linger)

builder.withConfig(jaasConfig).build()

}

/\*\*

\* Build the record to be published to Kafka from query, selections and response

\* @param query PipelineQuery

\* @param selectedCandidates Result after Selectors are executed

\* @param remainingCandidates Candidates which were not selected

\* @param droppedCandidates Candidates dropped during selection

\* @param response Result after Unmarshalling

\* @return A sequence of to-be-published ProducerRecords

\*/

def buildRecords(

query: Query,

selectedCandidates: Seq[CandidateWithDetails],

remainingCandidates: Seq[CandidateWithDetails],

droppedCandidates: Seq[CandidateWithDetails],

response: ResponseType

): Seq[ProducerRecord[K, V]]

final override def apply(

inputs: PipelineResultSideEffect.Inputs[Query, ResponseType]

): Stitch[Unit] = {

val records = buildRecords(

query = inputs.query,

selectedCandidates = inputs.selectedCandidates,

remainingCandidates = inputs.remainingCandidates,

droppedCandidates = inputs.droppedCandidates,

response = inputs.response

)

Stitch

.collect(

records

.map { record =>

Stitch.callFuture(kafkaProducer.send(record))

}

).unit

}

}