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

import com.twitter.app.Flag

import com.twitter.conversions.DurationOps.\_

import com.twitter.conversions.StorageUnitOps.\_

import com.twitter.decider.Decider

import com.twitter.decider.SimpleRecipient

import com.twitter.finatra.decider.modules.DeciderModule

import com.twitter.finatra.kafka.domain.AckMode

import com.twitter.finatra.kafka.domain.KafkaGroupId

import com.twitter.finatra.kafka.domain.KafkaTopic

import com.twitter.finatra.kafka.producers.FinagleKafkaProducerConfig

import com.twitter.finatra.kafka.producers.KafkaProducerConfig

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

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

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

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

import com.twitter.finatra.kafkastreams.config.KafkaStreamsConfig

import com.twitter.finatra.kafkastreams.config.SecureKafkaStreamsConfig

import com.twitter.finatra.kafkastreams.dsl.FinatraDslToCluster

import com.twitter.inject.TwitterModule

import com.twitter.unified\_user\_actions.enricher.driver.EnrichmentDriver

import com.twitter.unified\_user\_actions.enricher.hydrator.NoopHydrator

import com.twitter.unified\_user\_actions.enricher.internal.thriftscala.EnrichmentEnvelop

import com.twitter.unified\_user\_actions.enricher.internal.thriftscala.EnrichmentInstruction.NotificationTweetEnrichment

import com.twitter.unified\_user\_actions.enricher.internal.thriftscala.EnrichmentInstruction.TweetEnrichment

import com.twitter.unified\_user\_actions.enricher.internal.thriftscala.EnrichmentKey

import com.twitter.unified\_user\_actions.enricher.internal.thriftscala.EnrichmentPlan

import com.twitter.unified\_user\_actions.enricher.internal.thriftscala.EnrichmentStage

import com.twitter.unified\_user\_actions.enricher.internal.thriftscala.EnrichmentStageStatus

import com.twitter.unified\_user\_actions.enricher.internal.thriftscala.EnrichmentStageType

import com.twitter.unified\_user\_actions.enricher.partitioner.DefaultPartitioner

import com.twitter.unified\_user\_actions.enricher.partitioner.DefaultPartitioner.NullKey

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

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

import com.twitter.util.Await

import com.twitter.util.Future

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

import org.apache.kafka.streams.StreamsBuilder

import org.apache.kafka.streams.scala.kstream.Consumed

import org.apache.kafka.streams.scala.kstream.KStream

import org.apache.kafka.streams.scala.kstream.Produced

object EnrichmentPlannerServiceMain extends EnrichmentPlannerService {

val ApplicationId = "uua-enrichment-planner"

val InputTopic = "unified\_user\_actions"

val OutputPartitionedTopic = "unified\_user\_actions\_keyed\_dev"

val SamplingDecider = "EnrichmentPlannerSampling"

}

/\*\*

\* This service is the first step (planner) of the UUA Enrichment process.

\* It does the following:

\* 1. Read Prod UUA topic unified\_user\_actions from the Prod cluster and write to (see below) either Prod cluster (prod) or Dev cluster (dev/staging)

\* 2. For the write, it optionally randomly downsample the events when publishing, controlled by a Decider

\* 3. The output's key would be the first step of the repartitioning, most likely the EnrichmentKey of the Tweet type.

\*/

class EnrichmentPlannerService extends FinatraDslToCluster with SecureKafkaStreamsConfig {

import EnrichmentPlannerServiceMain.\_

val kafkaOutputCluster: Flag[String] = flag(

name = "kafka.output.server",

default = "",

help =

"""The output Kafka cluster.

|This is needed since we read from a cluster and potentially output to a different cluster.

|""".stripMargin

)

val kafkaOutputEnableTls: Flag[Boolean] = flag(

name = "kafka.output.enable.tls",

default = true,

help = ""

)

override val modules: Seq[TwitterModule] = Seq(

DeciderModule

)

override protected def configureKafkaStreams(builder: StreamsBuilder): Unit = {

val decider = injector.instance[Decider]

val driver = new EnrichmentDriver(

finalOutputTopic = NoopHydrator.OutputTopic,

partitionedTopic = OutputPartitionedTopic,

hydrator = new NoopHydrator,

partitioner = new DefaultPartitioner)

val builderWithoutOutput = builder.asScala

.stream(InputTopic)(Consumed.`with`(UnKeyedSerde, ScalaSerdes.Thrift[UnifiedUserAction]))

// this maps and filters out the nil envelop before further processing

.flatMapValues { uua =>

(uua.item match {

case Item.TweetInfo(\_) =>

Some(EnrichmentEnvelop(

envelopId = uua.hashCode.toLong,

uua = uua,

plan = EnrichmentPlan(Seq(

EnrichmentStage(

status = EnrichmentStageStatus.Initialized,

stageType = EnrichmentStageType.Repartition,

instructions = Seq(TweetEnrichment)

),

EnrichmentStage(

status = EnrichmentStageStatus.Initialized,

stageType = EnrichmentStageType.Hydration,

instructions = Seq(TweetEnrichment)

),

))

))

case Item.NotificationInfo(\_) =>

Some(EnrichmentEnvelop(

envelopId = uua.hashCode.toLong,

uua = uua,

plan = EnrichmentPlan(Seq(

EnrichmentStage(

status = EnrichmentStageStatus.Initialized,

stageType = EnrichmentStageType.Repartition,

instructions = Seq(NotificationTweetEnrichment)

),

EnrichmentStage(

status = EnrichmentStageStatus.Initialized,

stageType = EnrichmentStageType.Hydration,

instructions = Seq(NotificationTweetEnrichment)

),

))

))

case \_ => None

}).seq

}

// execute our driver logics

.flatMap((\_: UnKeyed, envelop: EnrichmentEnvelop) => {

// flatMap and Await.result is used here because our driver interface allows for

// both synchronous (repartition logic) and async operations (hydration logic), but in here

// we purely just need to repartition synchronously, and thus the flatMap + Await.result

// is used to simplify and make testing much easier.

val (keyOpt, value) = Await.result(driver.execute(NullKey, Future.value(envelop)))

keyOpt.map(key => (key, value)).seq

})

// then finally we sample based on the output keys

.filter((key, \_) =>

decider.isAvailable(feature = SamplingDecider, Some(SimpleRecipient(key.id))))

configureOutput(builderWithoutOutput)

}

private def configureOutput(kstream: KStream[EnrichmentKey, EnrichmentEnvelop]): Unit = {

if (kafkaOutputCluster().nonEmpty && kafkaOutputCluster() != bootstrapServer()) {

kstream.toCluster(

cluster = kafkaOutputCluster(),

topic = KafkaTopic(OutputPartitionedTopic),

clientId = s"$ApplicationId-output-producer",

kafkaProducerConfig =

if (kafkaOutputEnableTls())

FinagleKafkaProducerConfig[EnrichmentKey, EnrichmentEnvelop](kafkaProducerConfig =

KafkaProducerConfig(TwitterKafkaProducerConfig().requestTimeout(1.minute).configMap))

else

FinagleKafkaProducerConfig[EnrichmentKey, EnrichmentEnvelop](

kafkaProducerConfig = KafkaProducerConfig()

.requestTimeout(1.minute)),

statsReceiver = statsReceiver,

commitInterval = 15.seconds

)(Produced.`with`(ScalaSerdes.Thrift[EnrichmentKey], ScalaSerdes.Thrift[EnrichmentEnvelop]))

} else {

kstream.to(OutputPartitionedTopic)(

Produced.`with`(ScalaSerdes.Thrift[EnrichmentKey], ScalaSerdes.Thrift[EnrichmentEnvelop]))

}

}

override def streamsProperties(config: KafkaStreamsConfig): KafkaStreamsConfig = {

super

.streamsProperties(config)

.consumer.groupId(KafkaGroupId(ApplicationId))

.consumer.clientId(s"$ApplicationId-consumer")

.consumer.requestTimeout(30.seconds)

.consumer.sessionTimeout(30.seconds)

.consumer.fetchMin(1.megabyte)

.consumer.fetchMax(5.megabyte)

.consumer.receiveBuffer(32.megabytes)

.consumer.maxPollInterval(1.minute)

.consumer.maxPollRecords(50000)

.producer.clientId(s"$ApplicationId-producer")

.producer.batchSize(16.kilobytes)

.producer.bufferMemorySize(256.megabyte)

.producer.requestTimeout(30.seconds)

.producer.compressionType(CompressionType.LZ4)

.producer.ackMode(AckMode.ALL)

}

}