package com.twitter.recosinjector.event\_processors

import com.twitter.finagle.mtls.authentication.ServiceIdentifier

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.util.SnowflakeUtils

import com.twitter.gizmoduck.thriftscala.User

import com.twitter.recos.util.Action

import com.twitter.recos.util.Action.Action

import com.twitter.recosinjector.clients.Gizmoduck

import com.twitter.recosinjector.clients.SocialGraph

import com.twitter.recosinjector.clients.Tweetypie

import com.twitter.recosinjector.edges.TweetEventToUserTweetEntityGraphBuilder

import com.twitter.recosinjector.edges.TweetEventToUserUserGraphBuilder

import com.twitter.recosinjector.filters.TweetFilter

import com.twitter.recosinjector.filters.UserFilter

import com.twitter.recosinjector.publishers.KafkaEventPublisher

import com.twitter.recosinjector.util.TweetCreateEventDetails

import com.twitter.recosinjector.util.TweetDetails

import com.twitter.recosinjector.util.UserTweetEngagement

import com.twitter.scrooge.ThriftStructCodec

import com.twitter.tweetypie.thriftscala.Tweet

import com.twitter.tweetypie.thriftscala.TweetCreateEvent

import com.twitter.tweetypie.thriftscala.TweetEvent

import com.twitter.tweetypie.thriftscala.TweetEventData

import com.twitter.util.Future

/\*\*

\* Event processor for tweet\_events EventBus stream from Tweetypie. This stream provides all the

\* key events related to a new tweet, like Creation, Retweet, Quote Tweet, and Replying.

\* It also carries the entities/metadata information in a tweet, including

\* @ Mention, HashTag, MediaTag, URL, etc.

\*/

class TweetEventProcessor(

override val eventBusStreamName: String,

override val thriftStruct: ThriftStructCodec[TweetEvent],

override val serviceIdentifier: ServiceIdentifier,

userUserGraphMessageBuilder: TweetEventToUserUserGraphBuilder,

userUserGraphTopic: String,

userTweetEntityGraphMessageBuilder: TweetEventToUserTweetEntityGraphBuilder,

userTweetEntityGraphTopic: String,

kafkaEventPublisher: KafkaEventPublisher,

socialGraph: SocialGraph,

gizmoduck: Gizmoduck,

tweetypie: Tweetypie

)(

override implicit val statsReceiver: StatsReceiver)

extends EventBusProcessor[TweetEvent] {

private val tweetCreateEventCounter = statsReceiver.counter("num\_tweet\_create\_events")

private val nonTweetCreateEventCounter = statsReceiver.counter("num\_non\_tweet\_create\_events")

private val tweetActionStats = statsReceiver.scope("tweet\_action")

private val numUrlCounter = statsReceiver.counter("num\_tweet\_url")

private val numMediaUrlCounter = statsReceiver.counter("num\_tweet\_media\_url")

private val numHashTagCounter = statsReceiver.counter("num\_tweet\_hashtag")

private val numMentionsCounter = statsReceiver.counter("num\_tweet\_mention")

private val numMediatagCounter = statsReceiver.counter("num\_tweet\_mediatag")

private val numValidMentionsCounter = statsReceiver.counter("num\_tweet\_valid\_mention")

private val numValidMediatagCounter = statsReceiver.counter("num\_tweet\_valid\_mediatag")

private val numNullCastTweetCounter = statsReceiver.counter("num\_null\_cast\_tweet")

private val numNullCastSourceTweetCounter = statsReceiver.counter("num\_null\_cast\_source\_tweet")

private val numTweetFailSafetyLevelCounter = statsReceiver.counter("num\_fail\_tweetypie\_safety")

private val numAuthorUnsafeCounter = statsReceiver.counter("num\_author\_unsafe")

private val numProcessTweetCounter = statsReceiver.counter("num\_process\_tweet")

private val numNoProcessTweetCounter = statsReceiver.counter("num\_no\_process\_tweet")

private val selfRetweetCounter = statsReceiver.counter("num\_retweets\_self")

private val engageUserFilter = new UserFilter(gizmoduck)(statsReceiver.scope("author\_user"))

private val tweetFilter = new TweetFilter(tweetypie)

private def trackTweetCreateEventStats(details: TweetCreateEventDetails): Unit = {

tweetActionStats.counter(details.userTweetEngagement.action.toString).incr()

details.userTweetEngagement.tweetDetails.foreach { tweetDetails =>

tweetDetails.mentionUserIds.foreach(mention => numMentionsCounter.incr(mention.size))

tweetDetails.mediatagUserIds.foreach(mediatag => numMediatagCounter.incr(mediatag.size))

tweetDetails.urls.foreach(urls => numUrlCounter.incr(urls.size))

tweetDetails.mediaUrls.foreach(mediaUrls => numMediaUrlCounter.incr(mediaUrls.size))

tweetDetails.hashtags.foreach(hashtags => numHashTagCounter.incr(hashtags.size))

}

details.validMentionUserIds.foreach(mentions => numValidMentionsCounter.incr(mentions.size))

details.validMediatagUserIds.foreach(mediatags => numValidMediatagCounter.incr(mediatags.size))

}

/\*\*

\* Given a created tweet, return what type of tweet it is, i.e. Tweet, Retweet, Quote, or Reply。

\* Retweet, Quote, or Reply are responsive actions to a source tweet, so for these tweets,

\* we also return the tweet id and author of the source tweet (ex. the tweet being retweeted).

\*/

private def getTweetAction(tweetDetails: TweetDetails): Action = {

(tweetDetails.replySourceId, tweetDetails.retweetSourceId, tweetDetails.quoteSourceId) match {

case (Some(\_), \_, \_) =>

Action.Reply

case (\_, Some(\_), \_) =>

Action.Retweet

case (\_, \_, Some(\_)) =>

Action.Quote

case \_ =>

Action.Tweet

}

}

/\*\*

\* Given a list of mentioned users and mediatagged users in the tweet, return the users who

\* actually follow the source user.

\*/

private def getFollowedByIds(

sourceUserId: Long,

mentionUserIds: Option[Seq[Long]],

mediatagUserIds: Option[Seq[Long]]

): Future[Seq[Long]] = {

val uniqueEntityUserIds =

(mentionUserIds.getOrElse(Nil) ++ mediatagUserIds.getOrElse(Nil)).distinct

if (uniqueEntityUserIds.isEmpty) {

Future.Nil

} else {

socialGraph.followedByNotMutedBy(sourceUserId, uniqueEntityUserIds)

}

}

private def getSourceTweet(tweetDetails: TweetDetails): Future[Option[Tweet]] = {

tweetDetails.sourceTweetId match {

case Some(sourceTweetId) =>

tweetypie.getTweet(sourceTweetId)

case \_ =>

Future.None

}

}

/\*\*

\* Extract and return the details when the source user created a new tweet.

\*/

private def getTweetDetails(

tweet: Tweet,

engageUser: User

): Future[TweetCreateEventDetails] = {

val tweetDetails = TweetDetails(tweet)

val action = getTweetAction(tweetDetails)

val tweetCreationTimeMillis = SnowflakeUtils.tweetCreationTime(tweet.id).map(\_.inMilliseconds)

val engageUserId = engageUser.id

val userTweetEngagement = UserTweetEngagement(

engageUserId = engageUserId,

engageUser = Some(engageUser),

action = action,

engagementTimeMillis = tweetCreationTimeMillis,

tweetId = tweet.id,

tweetDetails = Some(tweetDetails)

)

val sourceTweetFut = getSourceTweet(tweetDetails)

val followedByIdsFut = getFollowedByIds(

engageUserId,

tweetDetails.mentionUserIds,

tweetDetails.mediatagUserIds

)

Future.join(followedByIdsFut, sourceTweetFut).map {

case (followedByIds, sourceTweet) =>

TweetCreateEventDetails(

userTweetEngagement = userTweetEngagement,

validEntityUserIds = followedByIds,

sourceTweetDetails = sourceTweet.map(TweetDetails)

)

}

}

/\*\*

\* Exclude any Retweets of one's own tweets

\*/

private def isEventSelfRetweet(tweetEvent: TweetCreateEventDetails): Boolean = {

(tweetEvent.userTweetEngagement.action == Action.Retweet) &&

tweetEvent.userTweetEngagement.tweetDetails.exists(

\_.sourceTweetUserId.contains(

tweetEvent.userTweetEngagement.engageUserId

))

}

private def isTweetPassSafetyFilter(tweetEvent: TweetCreateEventDetails): Future[Boolean] = {

tweetEvent.userTweetEngagement.action match {

case Action.Reply | Action.Retweet | Action.Quote =>

tweetEvent.userTweetEngagement.tweetDetails

.flatMap(\_.sourceTweetId).map { sourceTweetId =>

tweetFilter.filterForTweetypieSafetyLevel(sourceTweetId)

}.getOrElse(Future(false))

case Action.Tweet =>

tweetFilter.filterForTweetypieSafetyLevel(tweetEvent.userTweetEngagement.tweetId)

}

}

private def shouldProcessTweetEvent(event: TweetCreateEventDetails): Future[Boolean] = {

val engagement = event.userTweetEngagement

val engageUserId = engagement.engageUserId

val isNullCastTweet = engagement.tweetDetails.forall(\_.isNullCastTweet)

val isNullCastSourceTweet = event.sourceTweetDetails.exists(\_.isNullCastTweet)

val isSelfRetweet = isEventSelfRetweet(event)

val isEngageUserSafeFut = engageUserFilter.filterByUserId(engageUserId)

val isTweetPassSafetyFut = isTweetPassSafetyFilter(event)

Future.join(isEngageUserSafeFut, isTweetPassSafetyFut).map {

case (isEngageUserSafe, isTweetPassSafety) =>

if (isNullCastTweet) numNullCastTweetCounter.incr()

if (isNullCastSourceTweet) numNullCastSourceTweetCounter.incr()

if (!isEngageUserSafe) numAuthorUnsafeCounter.incr()

if (isSelfRetweet) selfRetweetCounter.incr()

if (!isTweetPassSafety) numTweetFailSafetyLevelCounter.incr()

!isNullCastTweet &&

!isNullCastSourceTweet &&

!isSelfRetweet &&

isEngageUserSafe &&

isTweetPassSafety

}

}

override def processEvent(event: TweetEvent): Future[Unit] = {

event.data match {

case TweetEventData.TweetCreateEvent(event: TweetCreateEvent) =>

getTweetDetails(

tweet = event.tweet,

engageUser = event.user

).flatMap { eventWithDetails =>

tweetCreateEventCounter.incr()

shouldProcessTweetEvent(eventWithDetails).map {

case true =>

numProcessTweetCounter.incr()

trackTweetCreateEventStats(eventWithDetails)

// Convert the event for UserUserGraph

userUserGraphMessageBuilder.processEvent(eventWithDetails).map { edges =>

edges.foreach { edge =>

kafkaEventPublisher.publish(edge.convertToRecosHoseMessage, userUserGraphTopic)

}

}

// Convert the event for UserTweetEntityGraph

userTweetEntityGraphMessageBuilder.processEvent(eventWithDetails).map { edges =>

edges.foreach { edge =>

kafkaEventPublisher

.publish(edge.convertToRecosHoseMessage, userTweetEntityGraphTopic)

}

}

case false =>

numNoProcessTweetCounter.incr()

}

}

case \_ =>

nonTweetCreateEventCounter.incr()

Future.Unit

}

}

}