package com.twitter.visibility.interfaces.conversations

import com.google.common.annotations.VisibleForTesting

import com.twitter.decider.Decider

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.gizmoduck.thriftscala.User

import com.twitter.spam.rtf.thriftscala.SafetyLevel

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.thriftscala.GetTweetFieldsResult

import com.twitter.tweetypie.thriftscala.TweetFieldsResultFound

import com.twitter.tweetypie.thriftscala.TweetFieldsResultState

import com.twitter.util.Stopwatch

import com.twitter.visibility.VisibilityLibrary

import com.twitter.visibility.common.filtered\_reason.FilteredReasonHelper

import com.twitter.visibility.models.ViewerContext

import com.twitter.visibility.rules.Interstitial

import com.twitter.visibility.rules.Tombstone

case class AdAvoidanceRequest(

conversationId: Long,

focalTweetId: Long,

tweets: Seq[(GetTweetFieldsResult, Option[SafetyLevel])],

authorMap: Map[

Long,

User

],

moderatedTweetIds: Seq[Long],

viewerContext: ViewerContext,

useRichText: Boolean = true)

case class AdAvoidanceResponse(dropAd: Map[Long, Boolean])

object AdAvoidanceLibrary {

type Type =

AdAvoidanceRequest => Stitch[AdAvoidanceResponse]

private def shouldAvoid(

result: TweetFieldsResultState,

tombstoneOpt: Option[VfTombstone],

statsReceiver: StatsReceiver

): Boolean = {

shouldAvoid(result, statsReceiver) || shouldAvoid(tombstoneOpt, statsReceiver)

}

private def shouldAvoid(

result: TweetFieldsResultState,

statsReceiver: StatsReceiver

): Boolean = {

result match {

case TweetFieldsResultState.Found(TweetFieldsResultFound(\_, \_, Some(filteredReason)))

if FilteredReasonHelper.isAvoid(filteredReason) =>

statsReceiver.counter("avoid").incr()

true

case \_ => false

}

}

private def shouldAvoid(

tombstoneOpt: Option[VfTombstone],

statsReceiver: StatsReceiver,

): Boolean = {

tombstoneOpt

.map(\_.action).collect {

case Tombstone(epitaph, \_) =>

statsReceiver.scope("tombstone").counter(epitaph.name).incr()

true

case interstitial: Interstitial =>

statsReceiver.scope("interstitial").counter(interstitial.reason.name).incr()

true

case \_ => false

}.getOrElse(false)

}

private def runTombstoneVisLib(

request: AdAvoidanceRequest,

tombstoneVisibilityLibrary: TombstoneVisibilityLibrary,

): Stitch[TombstoneVisibilityResponse] = {

val tombstoneRequest = TombstoneVisibilityRequest(

conversationId = request.conversationId,

focalTweetId = request.focalTweetId,

tweets = request.tweets,

authorMap = request.authorMap,

moderatedTweetIds = request.moderatedTweetIds,

viewerContext = request.viewerContext,

useRichText = request.useRichText

)

tombstoneVisibilityLibrary(tombstoneRequest)

}

def buildTweetAdAvoidanceMap(tweets: Seq[GetTweetFieldsResult]): Map[Long, Boolean] = tweets

.map(tweet => {

val shouldAvoid = tweet.tweetResult match {

case TweetFieldsResultState.Found(TweetFieldsResultFound(\_, \_, Some(filteredReason))) =>

FilteredReasonHelper.isAvoid(filteredReason)

case \_ => false

}

tweet.tweetId -> shouldAvoid

}).toMap

def apply(visibilityLibrary: VisibilityLibrary, decider: Decider): Type = {

val tvl =

TombstoneVisibilityLibrary(visibilityLibrary, visibilityLibrary.statsReceiver, decider)

buildLibrary(tvl, visibilityLibrary.statsReceiver)

}

@VisibleForTesting

def buildLibrary(

tvl: TombstoneVisibilityLibrary,

libraryStatsReceiver: StatsReceiver

): AdAvoidanceLibrary.Type = {

val statsReceiver = libraryStatsReceiver.scope("AdAvoidanceLibrary")

val reasonsStatsReceiver = statsReceiver.scope("reasons")

val latencyStatsReceiver = statsReceiver.scope("latency")

val vfLatencyOverallStat = latencyStatsReceiver.stat("vf\_latency\_overall")

val vfLatencyStitchBuildStat = latencyStatsReceiver.stat("vf\_latency\_stitch\_build")

val vfLatencyStitchRunStat = latencyStatsReceiver.stat("vf\_latency\_stitch\_run")

request: AdAvoidanceRequest => {

val elapsed = Stopwatch.start()

var runStitchStartMs = 0L

val tombstoneResponse: Stitch[TombstoneVisibilityResponse] =

runTombstoneVisLib(request, tvl)

val response = tombstoneResponse

.map({ response: TombstoneVisibilityResponse =>

statsReceiver.counter("requests").incr(request.tweets.size)

val dropResults: Seq[(Long, Boolean)] = request.tweets.map(tweetAndSafetyLevel => {

val tweet = tweetAndSafetyLevel.\_1

tweet.tweetId ->

shouldAvoid(

tweet.tweetResult,

response.tweetVerdicts.get(tweet.tweetId),

reasonsStatsReceiver)

})

AdAvoidanceResponse(dropAd = dropResults.toMap)

})

.onSuccess(\_ => {

val overallStatMs = elapsed().inMilliseconds

vfLatencyOverallStat.add(overallStatMs)

val runStitchEndMs = elapsed().inMilliseconds

vfLatencyStitchRunStat.add(runStitchEndMs - runStitchStartMs)

})

runStitchStartMs = elapsed().inMilliseconds

val buildStitchStatMs = elapsed().inMilliseconds

vfLatencyStitchBuildStat.add(buildStitchStatMs)

response

}

}

}