package com.twitter.visibility.builder

import com.twitter.datatools.entityservice.entities.thriftscala.FleetInterstitial

import com.twitter.decider.Decider

import com.twitter.decider.Decider.NullDecider

import com.twitter.finagle.stats.NullStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.logpipeline.client.common.EventPublisher

import com.twitter.logpipeline.client.EventPublisherManager

import com.twitter.logpipeline.client.serializers.EventLogMsgThriftStructSerializer

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

import com.twitter.visibility.builder.VerdictLogger.FailureCounterName

import com.twitter.visibility.builder.VerdictLogger.SuccessCounterName

import com.twitter.visibility.features.Feature

import com.twitter.visibility.logging.thriftscala.ActionSource

import com.twitter.visibility.logging.thriftscala.EntityId

import com.twitter.visibility.logging.thriftscala.EntityIdType

import com.twitter.visibility.logging.thriftscala.EntityIdValue

import com.twitter.visibility.logging.thriftscala.HealthActionType

import com.twitter.visibility.logging.thriftscala.MisinfoPolicyCategory

import com.twitter.visibility.logging.thriftscala.VFLibType

import com.twitter.visibility.logging.thriftscala.VFVerdictLogEntry

import com.twitter.visibility.models.ContentId

import com.twitter.visibility.rules.\_

object VerdictLogger {

private val BaseStatsNamespace = "vf\_verdict\_logger"

private val FailureCounterName = "failures"

private val SuccessCounterName = "successes"

val LogCategoryName: String = "visibility\_filtering\_verdicts"

val Empty: VerdictLogger = new VerdictLogger(NullStatsReceiver, NullDecider, None)

def apply(

statsReceiver: StatsReceiver,

decider: Decider

): VerdictLogger = {

val eventPublisher: EventPublisher[VFVerdictLogEntry] =

EventPublisherManager

.newScribePublisherBuilder(

LogCategoryName,

EventLogMsgThriftStructSerializer.getNewSerializer[VFVerdictLogEntry]()).build()

new VerdictLogger(statsReceiver.scope(BaseStatsNamespace), decider, Some(eventPublisher))

}

}

class VerdictLogger(

statsReceiver: StatsReceiver,

decider: Decider,

publisherOpt: Option[EventPublisher[VFVerdictLogEntry]]) {

def log(

verdictLogEntry: VFVerdictLogEntry,

publisher: EventPublisher[VFVerdictLogEntry]

): Unit = {

publisher

.publish(verdictLogEntry)

.onSuccess(\_ => statsReceiver.counter(SuccessCounterName).incr())

.onFailure { e =>

statsReceiver.counter(FailureCounterName).incr()

statsReceiver.scope(FailureCounterName).counter(e.getClass.getName).incr()

}

}

private def toEntityId(contentId: ContentId): Option[EntityId] = {

contentId match {

case ContentId.TweetId(id) => Some(EntityId(EntityIdType.TweetId, EntityIdValue.EntityId(id)))

case ContentId.UserId(id) => Some(EntityId(EntityIdType.UserId, EntityIdValue.EntityId(id)))

case ContentId.QuotedTweetRelationship(outerTweetId, \_) =>

Some(EntityId(EntityIdType.TweetId, EntityIdValue.EntityId(outerTweetId)))

case ContentId.NotificationId(Some(id)) =>

Some(EntityId(EntityIdType.NotificationId, EntityIdValue.EntityId(id)))

case ContentId.DmId(id) => Some(EntityId(EntityIdType.DmId, EntityIdValue.EntityId(id)))

case ContentId.BlenderTweetId(id) =>

Some(EntityId(EntityIdType.TweetId, EntityIdValue.EntityId(id)))

case ContentId.SpacePlusUserId(\_) =>

}

}

private def getLogEntryData(

actingRule: Option[Rule],

secondaryActingRules: Seq[Rule],

verdict: Action,

secondaryVerdicts: Seq[Action],

resolvedFeatureMap: Map[Feature[\_], Any]

): (Seq[String], Seq[ActionSource], Seq[HealthActionType], Option[FleetInterstitial]) = {

actingRule

.filter {

case decideredRule: DoesLogVerdictDecidered =>

decider.isAvailable(decideredRule.verdictLogDeciderKey.toString)

case rule: DoesLogVerdict => true

case \_ => false

}

.map { primaryRule =>

val secondaryRulesAndVerdicts = secondaryActingRules zip secondaryVerdicts

var actingRules: Seq[Rule] = Seq(primaryRule)

var actingRuleNames: Seq[String] = Seq(primaryRule.name)

var actionSources: Seq[ActionSource] = Seq()

var healthActionTypes: Seq[HealthActionType] = Seq(verdict.toHealthActionTypeThrift.get)

val misinfoPolicyCategory: Option[FleetInterstitial] = {

verdict match {

case softIntervention: SoftIntervention =>

softIntervention.fleetInterstitial

case tweetInterstitial: TweetInterstitial =>

tweetInterstitial.softIntervention.flatMap(\_.fleetInterstitial)

case \_ => None

}

}

secondaryRulesAndVerdicts.foreach(ruleAndVerdict => {

if (ruleAndVerdict.\_1.isInstanceOf[DoesLogVerdict]) {

actingRules = actingRules :+ ruleAndVerdict.\_1

actingRuleNames = actingRuleNames :+ ruleAndVerdict.\_1.name

healthActionTypes = healthActionTypes :+ ruleAndVerdict.\_2.toHealthActionTypeThrift.get

}

})

actingRules.foreach(rule => {

rule.actionSourceBuilder

.flatMap(\_.build(resolvedFeatureMap, verdict))

.map(actionSource => {

actionSources = actionSources :+ actionSource

})

})

(actingRuleNames, actionSources, healthActionTypes, misinfoPolicyCategory)

}

.getOrElse((Seq.empty[String], Seq.empty[ActionSource], Seq.empty[HealthActionType], None))

}

def scribeVerdict(

visibilityResult: VisibilityResult,

safetyLevel: SafetyLevel,

vfLibType: VFLibType,

viewerId: Option[Long] = None

): Unit = {

publisherOpt.foreach { publisher =>

toEntityId(visibilityResult.contentId).foreach { entityId =>

visibilityResult.verdict.toHealthActionTypeThrift.foreach { healthActionType =>

val (actioningRules, actionSources, healthActionTypes, misinfoPolicyCategory) =

getLogEntryData(

actingRule = visibilityResult.actingRule,

secondaryActingRules = visibilityResult.secondaryActingRules,

verdict = visibilityResult.verdict,

secondaryVerdicts = visibilityResult.secondaryVerdicts,

resolvedFeatureMap = visibilityResult.resolvedFeatureMap

)

if (actioningRules.nonEmpty) {

log(

VFVerdictLogEntry(

entityId = entityId,

viewerId = viewerId,

timestampMsec = System.currentTimeMillis(),

vfLibType = vfLibType,

healthActionType = healthActionType,

safetyLevel = safetyLevel,

actioningRules = actioningRules,

actionSources = actionSources,

healthActionTypes = healthActionTypes,

misinfoPolicyCategory =

fleetInterstitialToMisinfoPolicyCategory(misinfoPolicyCategory)

),

publisher

)

}

}

}

}

}

def fleetInterstitialToMisinfoPolicyCategory(

fleetInterstitialOption: Option[FleetInterstitial]

): Option[MisinfoPolicyCategory] = {

fleetInterstitialOption.map {

case FleetInterstitial.Generic =>

MisinfoPolicyCategory.Generic

case FleetInterstitial.Samm =>

MisinfoPolicyCategory.Samm

case FleetInterstitial.CivicIntegrity =>

MisinfoPolicyCategory.CivicIntegrity

case \_ => MisinfoPolicyCategory.Unknown

}

}

}