package com.twitter.visibility.engine

import com.twitter.finagle.stats.NullStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.stats.Verbosity

import com.twitter.servo.util.Gate

import com.twitter.servo.util.MemoizingStatsReceiver

import com.twitter.visibility.builder.VisibilityResult

import com.twitter.visibility.features.Feature

import com.twitter.visibility.models.SafetyLevel

import com.twitter.visibility.rules.NotEvaluated

import com.twitter.visibility.rules.RuleResult

import com.twitter.visibility.rules.State

import com.twitter.visibility.rules.State.Disabled

import com.twitter.visibility.rules.State.FeatureFailed

import com.twitter.visibility.rules.State.MissingFeature

import com.twitter.visibility.rules.State.RuleFailed

import com.twitter.visibility.rules.Action

case class VisibilityResultsMetricRecorder(

statsReceiver: StatsReceiver,

captureDebugStats: Gate[Unit]) {

private val scopedStatsReceiver = new MemoizingStatsReceiver(

statsReceiver.scope("visibility\_rule\_engine")

)

private val actionStats: StatsReceiver = scopedStatsReceiver.scope("by\_action")

private val featureFailureReceiver: StatsReceiver =

scopedStatsReceiver.scope("feature\_failed")

private val safetyLevelStatsReceiver: StatsReceiver =

scopedStatsReceiver.scope("from\_safety\_level")

private val ruleStatsReceiver: StatsReceiver = scopedStatsReceiver.scope("for\_rule")

private val ruleFailureReceiver: StatsReceiver =

scopedStatsReceiver.scope("rule\_failures")

private val failClosedReceiver: StatsReceiver =

scopedStatsReceiver.scope("fail\_closed")

private val ruleStatsBySafetyLevelReceiver: StatsReceiver =

scopedStatsReceiver.scope("for\_rule\_by\_safety\_level")

def recordSuccess(

safetyLevel: SafetyLevel,

result: VisibilityResult

): Unit = {

recordAction(safetyLevel, result.verdict.fullName)

val isFeatureFailure = result.ruleResultMap.values

.collectFirst {

case RuleResult(\_, FeatureFailed(\_)) =>

ruleFailureReceiver.counter("feature\_failed").incr()

true

}.getOrElse(false)

val isMissingFeature = result.ruleResultMap.values

.collectFirst {

case RuleResult(\_, MissingFeature(\_)) =>

ruleFailureReceiver.counter("missing\_feature").incr()

true

}.getOrElse(false)

val isRuleFailed = result.ruleResultMap.values

.collectFirst {

case RuleResult(\_, RuleFailed(\_)) =>

ruleFailureReceiver.counter("rule\_failed").incr()

true

}.getOrElse(false)

if (isFeatureFailure || isMissingFeature || isRuleFailed) {

ruleFailureReceiver.counter().incr()

}

if (captureDebugStats()) {

val ruleBySafetyLevelStat =

ruleStatsBySafetyLevelReceiver.scope(safetyLevel.name)

result.ruleResultMap.foreach {

case (rule, ruleResult) => {

ruleBySafetyLevelStat

.scope(rule.name)

.scope("action")

.counter(Verbosity.Debug, ruleResult.action.fullName).incr()

ruleBySafetyLevelStat

.scope(rule.name)

.scope("state")

.counter(Verbosity.Debug, ruleResult.state.name).incr()

}

}

}

}

def recordFailedFeature(

failedFeature: Feature[\_],

exception: Throwable

): Unit = {

featureFailureReceiver.counter().incr()

val featureStat = featureFailureReceiver.scope(failedFeature.name)

featureStat.counter().incr()

featureStat.counter(exception.getClass.getName).incr()

}

def recordAction(

safetyLevel: SafetyLevel,

action: String

): Unit = {

safetyLevelStatsReceiver.scope(safetyLevel.name).counter(action).incr()

actionStats.counter(action).incr()

}

def recordUnknownSafetyLevel(

safetyLevel: SafetyLevel

): Unit = {

safetyLevelStatsReceiver

.scope("unknown\_safety\_level")

.counter(safetyLevel.name.toLowerCase).incr()

}

def recordRuleMissingFeatures(

ruleName: String,

missingFeatures: Set[Feature[\_]]

): Unit = {

val ruleStat = ruleStatsReceiver.scope(ruleName)

missingFeatures.foreach { featureId =>

ruleStat.scope("missing\_feature").counter(featureId.name).incr()

}

ruleStat.scope("action").counter(NotEvaluated.fullName).incr()

ruleStat.scope("state").counter(MissingFeature(missingFeatures).name).incr()

}

def recordRuleFailedFeatures(

ruleName: String,

failedFeatures: Map[Feature[\_], Throwable]

): Unit = {

val ruleStat = ruleStatsReceiver.scope(ruleName)

ruleStat.scope("action").counter(NotEvaluated.fullName).incr()

ruleStat.scope("state").counter(FeatureFailed(failedFeatures).name).incr()

}

def recordFailClosed(rule: String, state: State) {

failClosedReceiver.scope(state.name).counter(rule).incr();

}

def recordRuleEvaluation(

ruleName: String,

action: Action,

state: State

): Unit = {

val ruleStat = ruleStatsReceiver.scope(ruleName)

ruleStat.scope("action").counter(action.fullName).incr()

ruleStat.scope("state").counter(state.name).incr()

}

def recordRuleFallbackAction(

ruleName: String

): Unit = {

val ruleStat = ruleStatsReceiver.scope(ruleName)

ruleStat.counter("fallback\_action").incr()

}

def recordRuleHoldBack(

ruleName: String

): Unit = {

ruleStatsReceiver.scope(ruleName).counter("heldback").incr()

}

def recordRuleFailed(

ruleName: String

): Unit = {

ruleStatsReceiver.scope(ruleName).counter("failed").incr()

}

def recordDisabledRule(

ruleName: String

): Unit = recordRuleEvaluation(ruleName, NotEvaluated, Disabled)

}

object NullVisibilityResultsMetricsRecorder

extends VisibilityResultsMetricRecorder(NullStatsReceiver, Gate.False)