package com.twitter.visibility.engine

import com.twitter.servo.util.Gate

import com.twitter.spam.rtf.thriftscala.{SafetyLevel => ThriftSafetyLevel}

import com.twitter.stitch.Stitch

import com.twitter.visibility.builder.VisibilityResult

import com.twitter.visibility.builder.VisibilityResultBuilder

import com.twitter.visibility.features.\_

import com.twitter.visibility.models.SafetyLevel

import com.twitter.visibility.models.SafetyLevel.DeprecatedSafetyLevel

import com.twitter.visibility.rules.EvaluationContext

import com.twitter.visibility.rules.State.\_

import com.twitter.visibility.rules.\_

import com.twitter.visibility.rules.providers.ProvidedEvaluationContext

import com.twitter.visibility.rules.providers.PolicyProvider

class VisibilityRuleEngine private[VisibilityRuleEngine] (

rulePreprocessor: VisibilityRulePreprocessor,

metricsRecorder: VisibilityResultsMetricRecorder,

enableComposableActions: Gate[Unit],

enableFailClosed: Gate[Unit],

policyProviderOpt: Option[PolicyProvider] = None)

extends DeciderableVisibilityRuleEngine {

private[visibility] def apply(

evaluationContext: ProvidedEvaluationContext,

visibilityPolicy: VisibilityPolicy,

visibilityResultBuilder: VisibilityResultBuilder,

enableShortCircuiting: Gate[Unit],

preprocessedRules: Option[Seq[Rule]]

): Stitch[VisibilityResult] = {

val (resultBuilder, rules) = preprocessedRules match {

case Some(r) =>

(visibilityResultBuilder, r)

case None =>

rulePreprocessor.evaluate(evaluationContext, visibilityPolicy, visibilityResultBuilder)

}

evaluate(evaluationContext, resultBuilder, rules, enableShortCircuiting)

}

def apply(

evaluationContext: EvaluationContext,

safetyLevel: SafetyLevel,

visibilityResultBuilder: VisibilityResultBuilder,

enableShortCircuiting: Gate[Unit] = Gate.True,

preprocessedRules: Option[Seq[Rule]] = None

): Stitch[VisibilityResult] = {

val visibilityPolicy = policyProviderOpt match {

case Some(policyProvider) =>

policyProvider.policyForSurface(safetyLevel)

case None => RuleBase.RuleMap(safetyLevel)

}

if (evaluationContext.params(safetyLevel.enabledParam)) {

apply(

ProvidedEvaluationContext.injectRuntimeRulesIntoEvaluationContext(

evaluationContext = evaluationContext,

safetyLevel = Some(safetyLevel),

policyProviderOpt = policyProviderOpt

),

visibilityPolicy,

visibilityResultBuilder,

enableShortCircuiting,

preprocessedRules

).onSuccess { result =>

metricsRecorder.recordSuccess(safetyLevel, result)

}

.onFailure { \_ =>

metricsRecorder.recordAction(safetyLevel, "failure")

}

} else {

metricsRecorder.recordAction(safetyLevel, "disabled")

val rules: Seq[Rule] = visibilityPolicy.forContentId(visibilityResultBuilder.contentId)

Stitch.value(

visibilityResultBuilder

.withRuleResultMap(rules.map(r => r -> RuleResult(Allow, Skipped)).toMap)

.withVerdict(verdict = Allow)

.withFinished(finished = true)

.build

)

}

}

def apply(

evaluationContext: EvaluationContext,

thriftSafetyLevel: ThriftSafetyLevel,

visibilityResultBuilder: VisibilityResultBuilder

): Stitch[VisibilityResult] = {

val safetyLevel: SafetyLevel = SafetyLevel.fromThrift(thriftSafetyLevel)

safetyLevel match {

case DeprecatedSafetyLevel =>

metricsRecorder.recordUnknownSafetyLevel(safetyLevel)

Stitch.value(

visibilityResultBuilder

.withVerdict(verdict = Allow)

.withFinished(finished = true)

.build

)

case thriftSafetyLevel: SafetyLevel =>

this(

ProvidedEvaluationContext.injectRuntimeRulesIntoEvaluationContext(

evaluationContext = evaluationContext,

safetyLevel = Some(safetyLevel),

policyProviderOpt = policyProviderOpt

),

thriftSafetyLevel,

visibilityResultBuilder

)

}

}

private[visibility] def evaluateRules(

evaluationContext: ProvidedEvaluationContext,

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

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

resultBuilderWithoutFailedFeatures: VisibilityResultBuilder,

preprocessedRules: Seq[Rule],

enableShortCircuiting: Gate[Unit]

): VisibilityResultBuilder = {

preprocessedRules

.foldLeft(resultBuilderWithoutFailedFeatures) { (builder, rule) =>

builder.ruleResults.get(rule) match {

case Some(RuleResult(\_, state)) if state == Evaluated || state == ShortCircuited =>

builder

case \_ =>

val failedFeatureDependencies: Map[Feature[\_], Throwable] =

failedFeatures.filterKeys(key => rule.featureDependencies.contains(key))

val shortCircuit =

builder.finished && enableShortCircuiting() &&

!(enableComposableActions() && builder.isVerdictComposable())

if (failedFeatureDependencies.nonEmpty && rule.fallbackActionBuilder.isEmpty) {

metricsRecorder.recordRuleFailedFeatures(rule.name, failedFeatureDependencies)

builder.withRuleResult(

rule,

RuleResult(NotEvaluated, FeatureFailed(failedFeatureDependencies)))

} else if (shortCircuit) {

metricsRecorder.recordRuleEvaluation(rule.name, NotEvaluated, ShortCircuited)

builder.withRuleResult(rule, RuleResult(builder.verdict, ShortCircuited))

} else {

if (failedFeatureDependencies.nonEmpty && rule.fallbackActionBuilder.nonEmpty) {

metricsRecorder.recordRuleFallbackAction(rule.name)

}

val ruleResult =

rule.evaluate(evaluationContext, resolvedFeatureMap)

metricsRecorder

.recordRuleEvaluation(rule.name, ruleResult.action, ruleResult.state)

val nextBuilder = (ruleResult.action, builder.finished) match {

case (NotEvaluated | Allow, \_) =>

ruleResult.state match {

case Heldback =>

metricsRecorder.recordRuleHoldBack(rule.name)

case RuleFailed(\_) =>

metricsRecorder.recordRuleFailed(rule.name)

case \_ =>

}

builder.withRuleResult(rule, ruleResult)

case (\_, true) =>

builder

.withRuleResult(rule, ruleResult)

.withSecondaryVerdict(ruleResult.action, rule)

case \_ =>

builder

.withRuleResult(rule, ruleResult)

.withVerdict(ruleResult.action, Some(rule))

.withFinished(true)

}

nextBuilder

}

}

}.withResolvedFeatureMap(resolvedFeatureMap)

}

private[visibility] def evaluateFailClosed(

evaluationContext: ProvidedEvaluationContext

): VisibilityResultBuilder => Stitch[VisibilityResultBuilder] = { builder =>

builder.failClosedException(evaluationContext) match {

case Some(e: FailClosedException) if enableFailClosed() =>

metricsRecorder.recordFailClosed(e.getRuleName, e.getState);

Stitch.exception(e)

case \_ => Stitch.value(builder)

}

}

private[visibility] def checkMarkFinished(

builder: VisibilityResultBuilder

): VisibilityResult = {

val allRulesEvaluated: Boolean = builder.ruleResults.values.forall {

case RuleResult(\_, state) =>

state == Evaluated || state == Disabled || state == Skipped

case \_ =>

false

}

if (allRulesEvaluated) {

builder.withFinished(true).build

} else {

builder.build

}

}

private[visibility] def evaluate(

evaluationContext: ProvidedEvaluationContext,

visibilityResultBuilder: VisibilityResultBuilder,

preprocessedRules: Seq[Rule],

enableShortCircuiting: Gate[Unit] = Gate.True

): Stitch[VisibilityResult] = {

val finalBuilder =

FeatureMap.resolve(visibilityResultBuilder.features, evaluationContext.statsReceiver).map {

resolvedFeatureMap =>

val (failedFeatureMap, successfulFeatureMap) = resolvedFeatureMap.constantMap.partition({

case (\_, \_: FeatureFailedPlaceholderObject) => true

case \_ => false

})

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

failedFeatureMap.mapValues({

case failurePlaceholder: FeatureFailedPlaceholderObject =>

failurePlaceholder.throwable

})

val resultBuilderWithoutFailedFeatures =

visibilityResultBuilder.withFeatureMap(ResolvedFeatureMap(successfulFeatureMap))

evaluateRules(

evaluationContext,

successfulFeatureMap,

failedFeatures,

resultBuilderWithoutFailedFeatures,

preprocessedRules,

enableShortCircuiting

)

}

finalBuilder.flatMap(evaluateFailClosed(evaluationContext)).map(checkMarkFinished)

}

}

object VisibilityRuleEngine {

def apply(

rulePreprocessor: Option[VisibilityRulePreprocessor] = None,

metricsRecorder: VisibilityResultsMetricRecorder = NullVisibilityResultsMetricsRecorder,

enableComposableActions: Gate[Unit] = Gate.False,

enableFailClosed: Gate[Unit] = Gate.False,

policyProviderOpt: Option[PolicyProvider] = None,

): VisibilityRuleEngine = {

new VisibilityRuleEngine(

rulePreprocessor.getOrElse(VisibilityRulePreprocessor(metricsRecorder)),

metricsRecorder,

enableComposableActions,

enableFailClosed,

policyProviderOpt = policyProviderOpt)

}

}