package com.twitter.visibility.engine

import com.twitter.abdecider.NullABDecider

import com.twitter.util.Return

import com.twitter.util.Throw

import com.twitter.util.Try

import com.twitter.visibility.builder.VisibilityResultBuilder

import com.twitter.visibility.features.\_

import com.twitter.visibility.models.SafetyLevel

import com.twitter.visibility.rules.Rule.DisabledRuleResult

import com.twitter.visibility.rules.Rule.EvaluatedRuleResult

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 VisibilityRulePreprocessor private (

metricsRecorder: VisibilityResultsMetricRecorder,

policyProviderOpt: Option[PolicyProvider] = None) {

private[engine] def filterEvaluableRules(

evaluationContext: ProvidedEvaluationContext,

resultBuilder: VisibilityResultBuilder,

rules: Seq[Rule]

): (VisibilityResultBuilder, Seq[Rule]) = {

val (builder, ruleList) = rules.foldLeft((resultBuilder, Seq.empty[Rule])) {

case ((builder, nextPassRules), rule) =>

if (evaluationContext.ruleEnabledInContext(rule)) {

val missingFeatures: Set[Feature[\_]] = rule.featureDependencies.collect {

case feature: Feature[\_] if !builder.featureMap.contains(feature) => feature

}

if (missingFeatures.isEmpty) {

(builder, nextPassRules :+ rule)

} else {

metricsRecorder.recordRuleMissingFeatures(rule.name, missingFeatures)

(

builder.withRuleResult(

rule,

RuleResult(NotEvaluated, MissingFeature(missingFeatures))

),

nextPassRules

)

}

} else {

(builder.withRuleResult(rule, DisabledRuleResult), nextPassRules)

}

}

(builder, ruleList)

}

private[visibility] def preFilterRules(

evaluationContext: ProvidedEvaluationContext,

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

resultBuilder: VisibilityResultBuilder,

rules: Seq[Rule]

): (VisibilityResultBuilder, Seq[Rule]) = {

val isResolvedFeatureMap = resultBuilder.featureMap.isInstanceOf[ResolvedFeatureMap]

val (builder, ruleList) = rules.foldLeft((resultBuilder, Seq.empty[Rule])) {

case ((builder, nextPassRules), rule) =>

rule.preFilter(evaluationContext, resolvedFeatureMap, NullABDecider) match {

case NeedsFullEvaluation =>

(builder, nextPassRules :+ rule)

case NotFiltered =>

(builder, nextPassRules :+ rule)

case Filtered if isResolvedFeatureMap =>

(builder, nextPassRules :+ rule)

case Filtered =>

(builder.withRuleResult(rule, EvaluatedRuleResult), nextPassRules)

}

}

(builder, ruleList)

}

private[visibility] def evaluate(

evaluationContext: ProvidedEvaluationContext,

safetyLevel: SafetyLevel,

resultBuilder: VisibilityResultBuilder

): (VisibilityResultBuilder, Seq[Rule]) = {

val visibilityPolicy = policyProviderOpt match {

case Some(policyProvider) =>

policyProvider.policyForSurface(safetyLevel)

case None => RuleBase.RuleMap(safetyLevel)

}

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

evaluate(evaluationContext, visibilityPolicy, resultBuilder)

} else {

metricsRecorder.recordAction(safetyLevel, "disabled")

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

val skippedResultBuilder = resultBuilder

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

.withVerdict(verdict = Allow)

.withFinished(finished = true)

(skippedResultBuilder, rules)

}

}

private[visibility] def evaluate(

evaluationContext: ProvidedEvaluationContext,

visibilityPolicy: VisibilityPolicy,

resultBuilder: VisibilityResultBuilder,

): (VisibilityResultBuilder, Seq[Rule]) = {

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

val (secondPassBuilder, secondPassRules) =

filterEvaluableRules(evaluationContext, resultBuilder, rules)

val secondPassFeatureMap = secondPassBuilder.featureMap

val secondPassConstantFeatures: Set[Feature[\_]] = RuleBase

.getFeaturesForRules(secondPassRules)

.filter(secondPassFeatureMap.containsConstant(\_))

val secondPassFeatureValues: Set[(Feature[\_], Any)] = secondPassConstantFeatures.map {

feature =>

Try(secondPassFeatureMap.getConstant(feature)) match {

case Return(value) => (feature, value)

case Throw(ex) =>

metricsRecorder.recordFailedFeature(feature, ex)

(feature, FeatureFailedPlaceholderObject(ex))

}

}

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

secondPassFeatureValues.filterNot {

case (\_, value) => value.isInstanceOf[FeatureFailedPlaceholderObject]

}.toMap

val (preFilteredResultBuilder, preFilteredRules) = preFilterRules(

evaluationContext,

resolvedFeatureMap,

secondPassBuilder,

secondPassRules

)

val preFilteredFeatureMap =

RuleBase.removeUnusedFeaturesFromFeatureMap(

preFilteredResultBuilder.featureMap,

preFilteredRules)

(preFilteredResultBuilder.withFeatureMap(preFilteredFeatureMap), preFilteredRules)

}

}

object VisibilityRulePreprocessor {

def apply(

metricsRecorder: VisibilityResultsMetricRecorder,

policyProviderOpt: Option[PolicyProvider] = None

): VisibilityRulePreprocessor = {

new VisibilityRulePreprocessor(metricsRecorder, policyProviderOpt)

}

}