package com.twitter.visibility

import com.twitter.abdecider.LoggingABDecider

import com.twitter.abdecider.NullABDecider

import com.twitter.decider.Decider

import com.twitter.decider.NullDecider

import com.twitter.featureswitches.v2.FeatureSwitches

import com.twitter.featureswitches.v2.NullFeatureSwitches

import com.twitter.finagle.stats.NullStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.logging.Logger

import com.twitter.logging.NullLogger

import com.twitter.servo.util.Gate

import com.twitter.servo.util.MemoizingStatsReceiver

import com.twitter.stitch.Stitch

import com.twitter.timelines.configapi.Params

import com.twitter.util.Try

import com.twitter.visibility.builder.\_

import com.twitter.visibility.common.stitch.StitchHelpers

import com.twitter.visibility.configapi.VisibilityParams

import com.twitter.visibility.configapi.configs.VisibilityDeciderGates

import com.twitter.visibility.engine.DeciderableVisibilityRuleEngine

import com.twitter.visibility.engine.VisibilityResultsMetricRecorder

import com.twitter.visibility.engine.VisibilityRuleEngine

import com.twitter.visibility.engine.VisibilityRulePreprocessor

import com.twitter.visibility.features.FeatureMap

import com.twitter.visibility.models.ContentId

import com.twitter.visibility.models.SafetyLevel

import com.twitter.visibility.models.ViewerContext

import com.twitter.visibility.rules.EvaluationContext

import com.twitter.visibility.rules.Rule

import com.twitter.visibility.rules.generators.TweetRuleGenerator

import com.twitter.visibility.rules.providers.InjectedPolicyProvider

import com.twitter.visibility.util.DeciderUtil

import com.twitter.visibility.util.FeatureSwitchUtil

import com.twitter.visibility.util.LoggingUtil

object VisibilityLibrary {

object Builder {

def apply(log: Logger, statsReceiver: StatsReceiver): Builder = new Builder(

log,

new MemoizingStatsReceiver(statsReceiver)

)

}

case class Builder(

log: Logger,

statsReceiver: StatsReceiver,

decider: Option[Decider] = None,

abDecider: Option[LoggingABDecider] = None,

featureSwitches: Option[FeatureSwitches] = None,

enableStitchProfiling: Gate[Unit] = Gate.False,

captureDebugStats: Gate[Unit] = Gate.False,

enableComposableActions: Gate[Unit] = Gate.False,

enableFailClosed: Gate[Unit] = Gate.False,

enableShortCircuiting: Gate[Unit] = Gate.True,

memoizeSafetyLevelParams: Gate[Unit] = Gate.False) {

def withDecider(decider: Decider): Builder = copy(decider = Some(decider))

@deprecated("use .withDecider and pass in a decider that is properly configured per DC")

def withDefaultDecider(isLocal: Boolean, useLocalOverrides: Boolean = false): Builder = {

if (isLocal) {

withLocalDecider

} else {

withDecider(

DeciderUtil.mkDecider(

useLocalDeciderOverrides = useLocalOverrides,

))

}

}

def withLocalDecider(): Builder = withDecider(DeciderUtil.mkLocalDecider)

def withNullDecider(): Builder =

withDecider(new NullDecider(isAvail = true, availabilityDefined = true))

def withABDecider(abDecider: LoggingABDecider, featureSwitches: FeatureSwitches): Builder =

abDecider match {

case abd: NullABDecider =>

copy(abDecider = Some(abd), featureSwitches = Some(NullFeatureSwitches))

case \_ =>

copy(

abDecider = Some(abDecider),

featureSwitches = Some(featureSwitches)

)

}

def withABDecider(abDecider: LoggingABDecider): Builder = abDecider match {

case abd: NullABDecider =>

withABDecider(abDecider = abd, featureSwitches = NullFeatureSwitches)

case \_ =>

withABDecider(

abDecider = abDecider,

featureSwitches =

FeatureSwitchUtil.mkVisibilityLibraryFeatureSwitches(abDecider, statsReceiver)

)

}

def withClientEventsLogger(clientEventsLogger: Logger): Builder =

withABDecider(DeciderUtil.mkABDecider(Some(clientEventsLogger)))

def withDefaultABDecider(isLocal: Boolean): Builder =

if (isLocal) {

withABDecider(NullABDecider)

} else {

withClientEventsLogger(LoggingUtil.mkDefaultLogger(statsReceiver))

}

def withNullABDecider(): Builder = withABDecider(NullABDecider)

def withEnableStitchProfiling(gate: Gate[Unit]): Builder =

copy(enableStitchProfiling = gate)

def withCaptureDebugStats(gate: Gate[Unit]): Builder =

copy(captureDebugStats = gate)

def withEnableComposableActions(gate: Gate[Unit]): Builder =

copy(enableComposableActions = gate)

def withEnableComposableActions(gateBoolean: Boolean): Builder = {

val gate = Gate.const(gateBoolean)

copy(enableComposableActions = gate)

}

def withEnableFailClosed(gate: Gate[Unit]): Builder =

copy(enableFailClosed = gate)

def withEnableFailClosed(gateBoolean: Boolean): Builder = {

val gate = Gate.const(gateBoolean)

copy(enableFailClosed = gate)

}

def withEnableShortCircuiting(gate: Gate[Unit]): Builder =

copy(enableShortCircuiting = gate)

def withEnableShortCircuiting(gateBoolean: Boolean): Builder = {

val gate = Gate.const(gateBoolean)

copy(enableShortCircuiting = gate)

}

def memoizeSafetyLevelParams(gate: Gate[Unit]): Builder =

copy(memoizeSafetyLevelParams = gate)

def memoizeSafetyLevelParams(gateBoolean: Boolean): Builder = {

val gate = Gate.const(gateBoolean)

copy(memoizeSafetyLevelParams = gate)

}

def build(): VisibilityLibrary = {

(decider, abDecider, featureSwitches) match {

case (None, \_, \_) =>

throw new IllegalStateException(

"Decider is unset! If intentional, please call .withNullDecider()."

)

case (\_, None, \_) =>

throw new IllegalStateException(

"ABDecider is unset! If intentional, please call .withNullABDecider()."

)

case (\_, \_, None) =>

throw new IllegalStateException(

"FeatureSwitches is unset! This is a bug."

)

case (Some(d), Some(abd), Some(fs)) =>

new VisibilityLibrary(

statsReceiver,

d,

abd,

VisibilityParams(log, statsReceiver, d, abd, fs),

enableStitchProfiling = enableStitchProfiling,

captureDebugStats = captureDebugStats,

enableComposableActions = enableComposableActions,

enableFailClosed = enableFailClosed,

enableShortCircuiting = enableShortCircuiting,

memoizeSafetyLevelParams = memoizeSafetyLevelParams)

}

}

}

val nullDecider = new NullDecider(true, true)

lazy val NullLibrary: VisibilityLibrary = new VisibilityLibrary(

NullStatsReceiver,

nullDecider,

NullABDecider,

VisibilityParams(

NullLogger,

NullStatsReceiver,

nullDecider,

NullABDecider,

NullFeatureSwitches),

enableStitchProfiling = Gate.False,

captureDebugStats = Gate.False,

enableComposableActions = Gate.False,

enableFailClosed = Gate.False,

enableShortCircuiting = Gate.True,

memoizeSafetyLevelParams = Gate.False

)

}

class VisibilityLibrary private[VisibilityLibrary] (

baseStatsReceiver: StatsReceiver,

decider: Decider,

abDecider: LoggingABDecider,

visibilityParams: VisibilityParams,

enableStitchProfiling: Gate[Unit],

captureDebugStats: Gate[Unit],

enableComposableActions: Gate[Unit],

enableFailClosed: Gate[Unit],

enableShortCircuiting: Gate[Unit],

memoizeSafetyLevelParams: Gate[Unit]) {

val statsReceiver: StatsReceiver =

new MemoizingStatsReceiver(baseStatsReceiver.scope("visibility\_library"))

val metricsRecorder = VisibilityResultsMetricRecorder(statsReceiver, captureDebugStats)

val visParams: VisibilityParams = visibilityParams

val visibilityDeciderGates = VisibilityDeciderGates(decider)

val profileStats: MemoizingStatsReceiver = new MemoizingStatsReceiver(

statsReceiver.scope("profiling"))

val perSafetyLevelProfileStats: StatsReceiver = profileStats.scope("for\_safety\_level")

val featureMapBuilder: FeatureMapBuilder.Build =

FeatureMapBuilder(statsReceiver, enableStitchProfiling)

private lazy val tweetRuleGenerator = new TweetRuleGenerator()

lazy val policyProvider = new InjectedPolicyProvider(

visibilityDeciderGates = visibilityDeciderGates,

tweetRuleGenerator = tweetRuleGenerator)

val candidateVisibilityRulePreprocessor: VisibilityRulePreprocessor = VisibilityRulePreprocessor(

metricsRecorder,

policyProviderOpt = Some(policyProvider)

)

val fallbackVisibilityRulePreprocessor: VisibilityRulePreprocessor = VisibilityRulePreprocessor(

metricsRecorder)

lazy val candidateVisibilityRuleEngine: VisibilityRuleEngine = VisibilityRuleEngine(

Some(candidateVisibilityRulePreprocessor),

metricsRecorder,

enableComposableActions,

enableFailClosed,

policyProviderOpt = Some(policyProvider)

)

lazy val fallbackVisibilityRuleEngine: VisibilityRuleEngine = VisibilityRuleEngine(

Some(fallbackVisibilityRulePreprocessor),

metricsRecorder,

enableComposableActions,

enableFailClosed)

val ruleEngineVersionStatsReceiver = statsReceiver.scope("rule\_engine\_version")

def isReleaseCandidateEnabled: Boolean = visibilityDeciderGates.enableExperimentalRuleEngine()

private def visibilityRuleEngine: DeciderableVisibilityRuleEngine = {

if (isReleaseCandidateEnabled) {

ruleEngineVersionStatsReceiver.counter("release\_candidate").incr()

candidateVisibilityRuleEngine

} else {

ruleEngineVersionStatsReceiver.counter("fallback").incr()

fallbackVisibilityRuleEngine

}

}

private def profileStitch[A](result: Stitch[A], safetyLevelName: String): Stitch[A] =

if (enableStitchProfiling()) {

StitchHelpers.profileStitch(

result,

Seq(profileStats, perSafetyLevelProfileStats.scope(safetyLevelName))

)

} else {

result

}

def getParams(viewerContext: ViewerContext, safetyLevel: SafetyLevel): Params = {

if (memoizeSafetyLevelParams()) {

visibilityParams.memoized(viewerContext, safetyLevel)

} else {

visibilityParams(viewerContext, safetyLevel)

}

}

def evaluationContextBuilder(viewerContext: ViewerContext): EvaluationContext.Builder =

EvaluationContext

.Builder(statsReceiver, visibilityParams, viewerContext)

.withMemoizedParams(memoizeSafetyLevelParams)

def runRuleEngine(

contentId: ContentId,

featureMap: FeatureMap,

evaluationContextBuilder: EvaluationContext.Builder,

safetyLevel: SafetyLevel

): Stitch[VisibilityResult] =

profileStitch(

visibilityRuleEngine(

evaluationContextBuilder.build(safetyLevel),

safetyLevel,

new VisibilityResultBuilder(contentId, featureMap),

enableShortCircuiting

),

safetyLevel.name

)

def runRuleEngine(

contentId: ContentId,

featureMap: FeatureMap,

viewerContext: ViewerContext,

safetyLevel: SafetyLevel

): Stitch[VisibilityResult] =

profileStitch(

visibilityRuleEngine(

EvaluationContext(safetyLevel, getParams(viewerContext, safetyLevel), statsReceiver),

safetyLevel,

new VisibilityResultBuilder(contentId, featureMap),

enableShortCircuiting

),

safetyLevel.name

)

def runRuleEngine(

viewerContext: ViewerContext,

safetyLevel: SafetyLevel,

preprocessedResultBuilder: VisibilityResultBuilder,

preprocessedRules: Seq[Rule]

): Stitch[VisibilityResult] =

profileStitch(

visibilityRuleEngine(

EvaluationContext(safetyLevel, getParams(viewerContext, safetyLevel), statsReceiver),

safetyLevel,

preprocessedResultBuilder,

enableShortCircuiting,

Some(preprocessedRules)

),

safetyLevel.name

)

def runRuleEngineBatch(

contentIds: Seq[ContentId],

featureMapProvider: (ContentId, SafetyLevel) => FeatureMap,

viewerContext: ViewerContext,

safetyLevel: SafetyLevel,

): Stitch[Seq[Try[VisibilityResult]]] = {

val params = getParams(viewerContext, safetyLevel)

profileStitch(

Stitch.traverse(contentIds) { contentId =>

visibilityRuleEngine(

EvaluationContext(safetyLevel, params, NullStatsReceiver),

safetyLevel,

new VisibilityResultBuilder(contentId, featureMapProvider(contentId, safetyLevel)),

enableShortCircuiting

).liftToTry

},

safetyLevel.name

)

}

def runRuleEngineBatch(

contentIds: Seq[ContentId],

featureMapProvider: (ContentId, SafetyLevel) => FeatureMap,

evaluationContextBuilder: EvaluationContext.Builder,

safetyLevel: SafetyLevel

): Stitch[Seq[Try[VisibilityResult]]] = {

val evaluationContext = evaluationContextBuilder.build(safetyLevel)

profileStitch(

Stitch.traverse(contentIds) { contentId =>

visibilityRuleEngine(

evaluationContext,

safetyLevel,

new VisibilityResultBuilder(contentId, featureMapProvider(contentId, safetyLevel)),

enableShortCircuiting

).liftToTry

},

safetyLevel.name

)

}

}