package com.twitter.visibility.interfaces.notifications

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.servo.util.Gate

import com.twitter.stitch.Stitch

import com.twitter.util.Throwables

import com.twitter.visibility.VisibilityLibrary

import com.twitter.visibility.builder.VisibilityResult

import com.twitter.visibility.builder.tweets.CommunityNotificationFeatures

import com.twitter.visibility.builder.tweets.UnmentionNotificationFeatures

import com.twitter.visibility.builder.users.AuthorDeviceFeatures

import com.twitter.visibility.builder.users.AuthorFeatures

import com.twitter.visibility.builder.users.RelationshipFeatures

import com.twitter.visibility.builder.users.ViewerAdvancedFilteringFeatures

import com.twitter.visibility.builder.users.ViewerFeatures

import com.twitter.visibility.common.UserDeviceSource

import com.twitter.visibility.common.UserRelationshipSource

import com.twitter.visibility.common.UserSource

import com.twitter.visibility.features.AuthorUserLabels

import com.twitter.visibility.features.Feature

import com.twitter.visibility.features.FeatureMap

import com.twitter.visibility.models.ViewerContext

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

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

import com.twitter.visibility.rules.Action

import com.twitter.visibility.rules.RuleResult

import com.twitter.visibility.rules.{Allow => AllowAction}

object NotificationsPlatformVisibilityLibrary {

type NotificationsPlatformVFType =

NotificationVFRequest => Stitch[NotificationsPlatformFilteringResponse]

private val AllowResponse: Stitch[NotificationsPlatformFilteringResponse] =

Stitch.value(AllowVerdict)

def apply(

userSource: UserSource,

userRelationshipSource: UserRelationshipSource,

userDeviceSource: UserDeviceSource,

visibilityLibrary: VisibilityLibrary,

enableShimFeatureHydration: Gate[Unit] = Gate.False

): NotificationsPlatformVFType = {

val libraryStatsReceiver = visibilityLibrary.statsReceiver

val vfEngineCounter = libraryStatsReceiver.counter("vf\_engine\_requests")

val authorFeatures = new AuthorFeatures(userSource, libraryStatsReceiver)

val authorDeviceFeatures = new AuthorDeviceFeatures(userDeviceSource, libraryStatsReceiver)

val viewerFeatures = new ViewerFeatures(userSource, libraryStatsReceiver)

val viewerAdvancedFilteringFeatures =

new ViewerAdvancedFilteringFeatures(userSource, libraryStatsReceiver)

val relationshipFeatures =

new RelationshipFeatures(userRelationshipSource, libraryStatsReceiver)

val isShimFeatureHydrationEnabled = enableShimFeatureHydration()

def runRuleEngine(candidate: NotificationVFRequest): Stitch[VisibilityResult] = {

val featureMap =

visibilityLibrary.featureMapBuilder(

Seq(

viewerFeatures.forViewerId(Some(candidate.recipientId)),

viewerAdvancedFilteringFeatures.forViewerId(Some(candidate.recipientId)),

authorFeatures.forAuthorId(candidate.subject.id),

authorDeviceFeatures.forAuthorId(candidate.subject.id),

relationshipFeatures.forAuthorId(candidate.subject.id, Some(candidate.recipientId)),

CommunityNotificationFeatures.ForNonCommunityTweetNotification,

UnmentionNotificationFeatures.ForNonUnmentionNotificationFeatures

)

)

vfEngineCounter.incr()

if (isShimFeatureHydrationEnabled) {

FeatureMap.resolve(featureMap, libraryStatsReceiver).flatMap { resolvedFeatureMap =>

visibilityLibrary.runRuleEngine(

contentId = candidate.subject,

featureMap = resolvedFeatureMap,

viewerContext =

ViewerContext.fromContextWithViewerIdFallback(Some(candidate.recipientId)),

safetyLevel = candidate.safetyLevel

)

}

} else {

visibilityLibrary.runRuleEngine(

contentId = candidate.subject,

featureMap = featureMap,

viewerContext =

ViewerContext.fromContextWithViewerIdFallback(Some(candidate.recipientId)),

safetyLevel = candidate.safetyLevel

)

}

}

{

case candidate: NotificationVFRequest =>

runRuleEngine(candidate).flatMap(failCloseForFailures(\_, libraryStatsReceiver))

case \_ =>

AllowResponse

}

}

private def failCloseForFailures(

visibilityResult: VisibilityResult,

stats: StatsReceiver

): Stitch[NotificationsPlatformFilteringResponse] = {

lazy val vfEngineSuccess = stats.counter("vf\_engine\_success")

lazy val vfEngineFailures = stats.counter("vf\_engine\_failures")

lazy val vfEngineFailuresMissing = stats.scope("vf\_engine\_failures").counter("missing")

lazy val vfEngineFailuresFailed = stats.scope("vf\_engine\_failures").counter("failed")

lazy val vfEngineFiltered = stats.counter("vf\_engine\_filtered")

val isFailedOrMissingFeature: RuleResult => Boolean = {

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

!(features.contains(AuthorUserLabels) && features.size == 1)

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

case \_ => false

}

val failedRuleResults =

visibilityResult.ruleResultMap.values.filter(isFailedOrMissingFeature(\_))

val (failedFeatures, missingFeatures) = failedRuleResults.partition {

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

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

case \_ => false

}

val failedOrMissingFeatures: Map[Feature[\_], String] = failedRuleResults

.collect {

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

features.map {

case (feature: Feature[\_], throwable: Throwable) =>

feature -> Throwables.mkString(throwable).mkString(" -> ")

}.toSet

case RuleResult(\_, MissingFeature(features)) => features.map(\_ -> "Feature missing.")

}.flatten.toMap

visibilityResult.verdict match {

case AllowAction if failedOrMissingFeatures.isEmpty =>

vfEngineSuccess.incr()

AllowResponse

case AllowAction if failedOrMissingFeatures.nonEmpty =>

vfEngineFailures.incr()

if (missingFeatures.nonEmpty) {

vfEngineFailuresMissing.incr()

}

if (failedFeatures.nonEmpty) {

vfEngineFailuresFailed.incr()

}

Stitch.value(FailedVerdict(failedOrMissingFeatures))

case action: Action =>

vfEngineFiltered.incr()

Stitch.value(FilteredVerdict(action))

}

}

}