package com.twitter.visibility.interfaces.notifications

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.notificationservice.model.notification.Notification

import com.twitter.notificationservice.model.notification.NotificationType

import com.twitter.notificationservice.model.notification.SimpleActivityNotification

import com.twitter.servo.util.Gate

import com.twitter.stitch.Stitch

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.TweetSource

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.FeatureMap

import com.twitter.visibility.models.ContentId.NotificationId

import com.twitter.visibility.models.SafetyLevel.NotificationsWriterV2

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 NotificationsVisibilityLibrary {

type Type = Notification => Stitch[NotificationsFilteringResponse]

private val AllowResponse: Stitch[NotificationsFilteringResponse] = Stitch.value(Allow)

def isApplicableOrganicNotificationType(notificationType: NotificationType): Boolean = {

NotificationType.isTlsActivityType(notificationType) ||

NotificationType.isReactionType(notificationType)

}

def apply(

visibilityLibrary: VisibilityLibrary,

userSource: UserSource,

userRelationshipSource: UserRelationshipSource,

userDeviceSource: UserDeviceSource,

tweetSource: TweetSource,

enableShimFeatureHydration: Gate[Unit] = Gate.False,

enableCommunityTweetHydration: Gate[Long] = Gate.False,

enableUnmentionHydration: Gate[Long] = Gate.False,

): Type = {

val libraryStatsReceiver = visibilityLibrary.statsReceiver

lazy 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 communityNotificationFeatures =

new CommunityNotificationFeatures(

tweetSource,

enableCommunityTweetHydration,

libraryStatsReceiver)

val unmentionNotificationFeatures = new UnmentionNotificationFeatures(

tweetSource = tweetSource,

enableUnmentionHydration = enableUnmentionHydration,

statsReceiver = libraryStatsReceiver

)

val viewerAdvancedFilteringFeatures =

new ViewerAdvancedFilteringFeatures(userSource, libraryStatsReceiver)

val relationshipFeatures =

new RelationshipFeatures(userRelationshipSource, libraryStatsReceiver)

val isShimFeatureHydrationEnabled = enableShimFeatureHydration()

def runRuleEngine(

visibilityLibrary: VisibilityLibrary,

candidate: Notification

): Stitch[VisibilityResult] = {

candidate match {

case notification: SimpleActivityNotification[\_] =>

vfEngineCounter.incr()

val featureMap = visibilityLibrary.featureMapBuilder(

Seq(

viewerFeatures.forViewerId(Some(notification.target)),

viewerAdvancedFilteringFeatures.forViewerId(Some(notification.target)),

authorFeatures.forAuthorId(notification.subjectId),

authorDeviceFeatures.forAuthorId(notification.subjectId),

relationshipFeatures

.forAuthorId(notification.subjectId, Some(notification.target)),

communityNotificationFeatures.forNotification(notification),

unmentionNotificationFeatures.forNotification(notification)

)

)

if (isShimFeatureHydrationEnabled) {

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

visibilityLibrary.runRuleEngine(

contentId =

featureMap = resolvedFeatureMap,

viewerContext =

ViewerContext.fromContextWithViewerIdFallback(Some(notification.target)),

safetyLevel = NotificationsWriterV2

)

}

} else {

visibilityLibrary.runRuleEngine(

contentId = NotificationId(tweetId = None),

featureMap = featureMap,

viewerContext =

ViewerContext.fromContextWithViewerIdFallback(Some(notification.target)),

safetyLevel = NotificationsWriterV2

)

}

}

}

{

case candidate if isApplicableOrganicNotificationType(candidate.notificationType) =>

runRuleEngine(visibilityLibrary, candidate)

.flatMap(failCloseForFailures(\_, libraryStatsReceiver))

case \_ =>

AllowResponse

}

}

def failCloseForFailures(

visibilityResult: VisibilityResult,

stats: StatsReceiver

): Stitch[NotificationsFilteringResponse] = {

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 = failedRuleResults

.collect {

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

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

}.toSet.flatten

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(Failed(failedOrMissingFeatures))

case action: Action =>

vfEngineFiltered.incr()

Stitch.value(Filtered(action))

}

}

}