package com.twitter.visibility.interfaces.spaces

import com.twitter.servo.util.Gate

import com.twitter.stitch.Stitch

import com.twitter.strato.client.{Client => StratoClient}

import com.twitter.visibility.VisibilityLibrary

import com.twitter.visibility.builder.VisibilityResult

import com.twitter.visibility.builder.common.MutedKeywordFeatures

import com.twitter.visibility.builder.spaces.SpaceFeatures

import com.twitter.visibility.builder.spaces.StratoSpaceLabelMaps

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

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

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

import com.twitter.visibility.common.\_

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

import com.twitter.visibility.features.FeatureMap

import com.twitter.visibility.models.ContentId.SpaceId

import com.twitter.visibility.models.ContentId.SpacePlusUserId

import com.twitter.visibility.rules.EvaluationContext

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

import com.twitter.visibility.rules.utils.ShimUtils

object SpaceVisibilityLibrary {

type Type = SpaceVisibilityRequest => Stitch[VisibilityResult]

def apply(

visibilityLibrary: VisibilityLibrary,

stratoClient: StratoClient,

userSource: UserSource,

userRelationshipSource: UserRelationshipSource,

enableVfFeatureHydrationSpaceShim: Gate[Unit] = Gate.False

): Type = {

val libraryStatsReceiver = visibilityLibrary.statsReceiver

val stratoClientStatsReceiver = visibilityLibrary.statsReceiver.scope("strato")

val vfLatencyStatsReceiver = visibilityLibrary.statsReceiver.scope("vf\_latency")

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

val spaceLabelMaps = new StratoSpaceLabelMaps(

SpaceSafetyLabelMapSource.fromStrato(stratoClient, stratoClientStatsReceiver),

libraryStatsReceiver)

val audioSpaceSource = AudioSpaceSource.fromStrato(stratoClient, stratoClientStatsReceiver)

val viewerFeatures = new ViewerFeatures(userSource, libraryStatsReceiver)

val authorFeatures = new AuthorFeatures(userSource, libraryStatsReceiver)

val relationshipFeatures =

new RelationshipFeatures(userRelationshipSource, libraryStatsReceiver)

val mutedKeywordFeatures = new MutedKeywordFeatures(

userSource,

userRelationshipSource,

KeywordMatcher.matcher(libraryStatsReceiver),

libraryStatsReceiver,

Gate.False

)

val spaceFeatures =

new SpaceFeatures(

spaceLabelMaps,

authorFeatures,

relationshipFeatures,

mutedKeywordFeatures,

audioSpaceSource)

{ r: SpaceVisibilityRequest =>

vfEngineCounter.incr()

val isVfFeatureHydrationEnabled = enableVfFeatureHydrationSpaceShim()

val viewerId = r.viewerContext.userId

val authorIds: Option[Seq[Long]] = r.spaceHostAndAdminUserIds

val contentId = {

(viewerId, authorIds) match {

case (Some(viewer), Some(authors)) if authors.contains(viewer) => SpaceId(r.spaceId)

case \_ => SpacePlusUserId(r.spaceId)

}

}

val featureMap =

visibilityLibrary.featureMapBuilder(

Seq(

spaceFeatures.forSpaceAndAuthorIds(r.spaceId, viewerId, authorIds),

viewerFeatures.forViewerContext(r.viewerContext),

)

)

val resp = if (isVfFeatureHydrationEnabled) {

val evaluationContext = ProvidedEvaluationContext.injectRuntimeRulesIntoEvaluationContext(

evaluationContext = EvaluationContext(

r.safetyLevel,

visibilityLibrary.getParams(r.viewerContext, r.safetyLevel),

visibilityLibrary.statsReceiver)

)

val preFilteredFeatureMap =

ShimUtils.preFilterFeatureMap(featureMap, r.safetyLevel, contentId, evaluationContext)

FeatureMap

.resolve(preFilteredFeatureMap, libraryStatsReceiver).flatMap { resolvedFeatureMap =>

visibilityLibrary

.runRuleEngine(

contentId,

resolvedFeatureMap,

r.viewerContext,

r.safetyLevel

)

}

} else {

visibilityLibrary

.runRuleEngine(

contentId,

featureMap,

r.viewerContext,

r.safetyLevel

)

}

StitchHelpers.profileStitch(resp, Seq(vfLatencyStatsReceiver))

}

}

}