package com.twitter.visibility.interfaces.push\_service

import com.twitter.gizmoduck.thriftscala.User

import com.twitter.servo.util.Gate

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.stitch.Stitch

import com.twitter.stitch.tweetypie.TweetyPie.TweetyPieResult

import com.twitter.storehaus.ReadableStore

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

import com.twitter.tweetypie.thriftscala.Tweet

import com.twitter.visibility.VisibilityLibrary

import com.twitter.visibility.builder.tweets.TweetFeatures

import com.twitter.visibility.builder.tweets.StratoTweetLabelMaps

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.builder.VisibilityResult

import com.twitter.visibility.common.\_

import com.twitter.visibility.common.UserRelationshipSource

import com.twitter.visibility.common.UserSource

import com.twitter.visibility.features.FeatureMap

import com.twitter.visibility.features.TweetIsInnerQuotedTweet

import com.twitter.visibility.features.TweetIsRetweet

import com.twitter.visibility.features.TweetIsSourceTweet

import com.twitter.visibility.interfaces.push\_service.PushServiceVisibilityLibraryUtil.\_

import com.twitter.visibility.models.ContentId

import com.twitter.visibility.models.ViewerContext

object TweetType extends Enumeration {

type TweetType = Value

val ORIGINAL, SOURCE, QUOTED = Value

}

import com.twitter.visibility.interfaces.push\_service.TweetType.\_

object PushServiceVisibilityLibrary {

type Type = PushServiceVisibilityRequest => Stitch[PushServiceVisibilityResponse]

def apply(

visibilityLibrary: VisibilityLibrary,

userSource: UserSource,

userRelationshipSource: UserRelationshipSource,

stratoClient: StratoClient,

enableParityTest: Gate[Unit] = Gate.False,

cachedTweetyPieStoreV2: ReadableStore[Long, TweetyPieResult] = ReadableStore.empty,

safeCachedTweetyPieStoreV2: ReadableStore[Long, TweetyPieResult] = ReadableStore.empty,

)(

implicit statsReceiver: StatsReceiver

): Type = {

val stats = statsReceiver.scope("push\_service\_vf")

val candidateTweetCounter = stats.counter("request\_cnt")

val allowedTweetCounter = stats.counter("allow\_cnt")

val droppedTweetCounter = stats.counter("drop\_cnt")

val failedTweetCounter = stats.counter("fail\_cnt")

val authorLabelsEmptyCount = stats.counter("author\_labels\_empty\_cnt")

val authorLabelsCount = stats.counter("author\_labels\_cnt")

val tweetLabelMaps = new StratoTweetLabelMaps(

SafetyLabelMapSource.fromSafetyLabelMapFetcher(

PushServiceSafetyLabelMapFetcher(stratoClient, stats)))

val viewerFeatures = new ViewerFeatures(UserSource.empty, stats)

val tweetFeatures = new TweetFeatures(tweetLabelMaps, stats)

val authorFeatures = new AuthorFeatures(userSource, stats)

val relationshipFeatures = new RelationshipFeatures(UserRelationshipSource.empty, stats)

val parityTester = new PushServiceVisibilityLibraryParity(

cachedTweetyPieStoreV2,

safeCachedTweetyPieStoreV2

)(statsReceiver)

def buildFeatureMap(

request: PushServiceVisibilityRequest,

tweet: Tweet,

tweetType: TweetType,

author: Option[User] = None,

): FeatureMap = {

val authorId = author.map(\_.id) orElse getAuthorId(tweet)

(author.map(authorFeatures.forAuthor(\_)) orElse

getAuthorId(tweet).map(authorFeatures.forAuthorId(\_))) match {

case Some(authorVisibilityFeatures) =>

visibilityLibrary.featureMapBuilder(

Seq(

viewerFeatures.forViewerContext(ViewerContext.fromContextWithViewerIdFallback(None)),

tweetFeatures.forTweet(tweet),

authorVisibilityFeatures,

relationshipFeatures.forAuthorId(authorId.get, None),

\_.withConstantFeature(TweetIsInnerQuotedTweet, tweetType == QUOTED),

\_.withConstantFeature(TweetIsRetweet, request.isRetweet),

\_.withConstantFeature(TweetIsSourceTweet, tweetType == SOURCE)

)

)

case \_ =>

visibilityLibrary.featureMapBuilder(

Seq(

viewerFeatures.forViewerContext(ViewerContext.fromContextWithViewerIdFallback(None)),

tweetFeatures.forTweet(tweet),

\_.withConstantFeature(TweetIsInnerQuotedTweet, tweetType == QUOTED),

\_.withConstantFeature(TweetIsRetweet, request.isRetweet),

\_.withConstantFeature(TweetIsSourceTweet, tweetType == SOURCE)

)

)

}

}

def runRuleEngineForTweet(

request: PushServiceVisibilityRequest,

tweet: Tweet,

tweetType: TweetType,

author: Option[User] = None,

): Stitch[VisibilityResult] = {

val featureMap = buildFeatureMap(request, tweet, tweetType, author)

val contentId = ContentId.TweetId(tweet.id)

visibilityLibrary.runRuleEngine(

contentId,

featureMap,

request.viewerContext,

request.safetyLevel)

}

def runRuleEngineForAuthor(

request: PushServiceVisibilityRequest,

tweet: Tweet,

tweetType: TweetType,

author: Option[User] = None,

): Stitch[VisibilityResult] = {

val featureMap = buildFeatureMap(request, tweet, tweetType, author)

val authorId = author.map(\_.id).getOrElse(getAuthorId(tweet).get)

val contentId = ContentId.UserId(authorId)

visibilityLibrary.runRuleEngine(

contentId,

featureMap,

request.viewerContext,

request.safetyLevel)

}

def getAllVisibilityFilters(

request: PushServiceVisibilityRequest

): Stitch[PushServiceVisibilityResponse] = {

val tweetResult =

runRuleEngineForTweet(request, request.tweet, ORIGINAL, Some(request.author))

val authorResult =

runRuleEngineForAuthor(request, request.tweet, ORIGINAL, Some(request.author))

val sourceTweetResult = request.sourceTweet

.map(runRuleEngineForTweet(request, \_, SOURCE).map(Some(\_))).getOrElse(Stitch.None)

val quotedTweetResult = request.quotedTweet

.map(runRuleEngineForTweet(request, \_, QUOTED).map(Some(\_))).getOrElse(Stitch.None)

Stitch.join(tweetResult, authorResult, sourceTweetResult, quotedTweetResult).map {

case (tweetResult, authorResult, sourceTweetResult, quotedTweetResult) =>

PushServiceVisibilityResponse(

tweetResult,

authorResult,

sourceTweetResult,

quotedTweetResult)

}

}

{ request: PushServiceVisibilityRequest =>

candidateTweetCounter.incr()

request.author.labels match {

case Some(labels) if (!labels.\_1.isEmpty) => authorLabelsCount.incr()

case \_ => authorLabelsEmptyCount.incr()

}

val response = getAllVisibilityFilters(request)

.onSuccess { response =>

if (response.shouldAllow) allowedTweetCounter.incr() else droppedTweetCounter.incr()

}.onFailure { \_ => failedTweetCounter.incr() }

if (enableParityTest()) {

response.applyEffect { resp => Stitch.async(parityTester.runParityTest(request, resp)) }

} else {

response

}

}

}

}