package com.twitter.follow\_recommendations.common.predicates.gizmoduck

import com.twitter.decider.Decider

import com.twitter.decider.RandomRecipient

import com.twitter.escherbird.util.stitchcache.StitchCache

import com.twitter.finagle.Memcached.Client

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.util.DefaultTimer

import com.twitter.follow\_recommendations.common.base.StatsUtil

import com.twitter.follow\_recommendations.common.base.Predicate

import com.twitter.follow\_recommendations.common.base.PredicateResult

import com.twitter.follow\_recommendations.common.clients.cache.MemcacheClient

import com.twitter.follow\_recommendations.common.clients.cache.ThriftBijection

import com.twitter.follow\_recommendations.common.models.FilterReason.\_

import com.twitter.follow\_recommendations.common.models.AddressBookMetadata

import com.twitter.follow\_recommendations.common.models.CandidateUser

import com.twitter.follow\_recommendations.common.models.FilterReason

import com.twitter.follow\_recommendations.common.predicates.gizmoduck.GizmoduckPredicate.\_

import com.twitter.follow\_recommendations.common.predicates.gizmoduck.GizmoduckPredicateParams.\_

import com.twitter.follow\_recommendations.configapi.deciders.DeciderKey

import com.twitter.gizmoduck.thriftscala.LabelValue.BlinkBad

import com.twitter.gizmoduck.thriftscala.LabelValue.BlinkWorst

import com.twitter.gizmoduck.thriftscala.LabelValue

import com.twitter.gizmoduck.thriftscala.LookupContext

import com.twitter.gizmoduck.thriftscala.QueryFields

import com.twitter.gizmoduck.thriftscala.User

import com.twitter.gizmoduck.thriftscala.UserResult

import com.twitter.product\_mixer.core.model.marshalling.request.HasClientContext

import com.twitter.scrooge.CompactThriftSerializer

import com.twitter.spam.rtf.thriftscala.SafetyLevel

import com.twitter.stitch.Stitch

import com.twitter.stitch.gizmoduck.Gizmoduck

import com.twitter.timelines.configapi.HasParams

import com.twitter.util.Duration

import com.twitter.util.logging.Logging

import java.lang.{Long => JLong}

import javax.inject.Inject

import javax.inject.Singleton

/\*\*

\* In this filter, we want to check 4 categories of conditions:

\* - if candidate is discoverable given that it's from an address-book/phone-book based source

\* - if candidate is unsuitable based on it's safety sub-fields in gizmoduck

\* - if candidate is withheld because of country-specific take-down policies

\* - if candidate is marked as bad/worst based on blink labels

\* We fail close on the query as this is a product-critical filter

\*/

@Singleton

case class GizmoduckPredicate @Inject() (

gizmoduck: Gizmoduck,

client: Client,

statsReceiver: StatsReceiver,

decider: Decider = Decider.False)

extends Predicate[(HasClientContext with HasParams, CandidateUser)]

with Logging {

private val stats: StatsReceiver = statsReceiver.scope(this.getClass.getName)

// track # of Gizmoduck predicate queries that yielded valid & invalid predicate results

private val validPredicateResultCounter = stats.counter("predicate\_valid")

private val invalidPredicateResultCounter = stats.counter("predicate\_invalid")

// track # of cases where no Gizmoduck user was found

private val noGizmoduckUserCounter = stats.counter("no\_gizmoduck\_user\_found")

private val gizmoduckCache = StitchCache[JLong, UserResult](

maxCacheSize = MaxCacheSize,

ttl = CacheTTL,

statsReceiver = stats.scope("cache"),

underlyingCall = getByUserId

)

// Distributed Twemcache to store UserResult objects keyed on user IDs

val bijection = new ThriftBijection[UserResult] {

override val serializer = CompactThriftSerializer(UserResult)

}

val memcacheClient = MemcacheClient[UserResult](

client = client,

dest = "/s/cache/frs:twemcaches",

valueBijection = bijection,

ttl = CacheTTL,

statsReceiver = stats.scope("twemcache")

)

// main method used to apply GizmoduckPredicate to a candidate user

override def apply(

pair: (HasClientContext with HasParams, CandidateUser)

): Stitch[PredicateResult] = {

val (request, candidate) = pair

// measure the latency of the getGizmoduckPredicateResult, since this predicate

// check is product-critical and relies on querying a core service (Gizmoduck)

StatsUtil.profileStitch(

getGizmoduckPredicateResult(request, candidate),

stats.scope("getGizmoduckPredicateResult")

)

}

private def getGizmoduckPredicateResult(

request: HasClientContext with HasParams,

candidate: CandidateUser

): Stitch[PredicateResult] = {

val timeout: Duration = request.params(GizmoduckGetTimeout)

val deciderKey: String = DeciderKey.EnableGizmoduckCaching.toString

val enableDistributedCaching: Boolean = decider.isAvailable(deciderKey, Some(RandomRecipient))

// try getting an existing UserResult from cache if possible

val userResultStitch: Stitch[UserResult] =

enableDistributedCaching match {

// read from memcache

case true => memcacheClient.readThrough(

// add a key prefix to address cache key collisions

key = "GizmoduckPredicate" + candidate.id.toString,

underlyingCall = () => getByUserId(candidate.id)

)

// read from local cache

case false => gizmoduckCache.readThrough(candidate.id)

}

val predicateResultStitch = userResultStitch.map {

userResult => {

val predicateResult = getPredicateResult(request, candidate, userResult)

if (enableDistributedCaching) {

predicateResult match {

case PredicateResult.Valid =>

stats.scope("twemcache").counter("predicate\_valid").incr()

case PredicateResult.Invalid(reasons) =>

stats.scope("twemcache").counter("predicate\_invalid").incr()

}

// log metrics to check if local cache value matches distributed cache value

logPredicateResultEquality(

request,

candidate,

predicateResult

)

} else {

predicateResult match {

case PredicateResult.Valid =>

stats.scope("cache").counter("predicate\_valid").incr()

case PredicateResult.Invalid(reasons) =>

stats.scope("cache").counter("predicate\_invalid").incr()

}

}

predicateResult

}

}

predicateResultStitch

.within(timeout)(DefaultTimer)

.rescue { // fail-open when timeout or exception

case e: Exception =>

stats.scope("rescued").counter(e.getClass.getSimpleName).incr()

invalidPredicateResultCounter.incr()

Stitch(PredicateResult.Invalid(Set(FailOpen)))

}

}

private def logPredicateResultEquality(

request: HasClientContext with HasParams,

candidate: CandidateUser,

predicateResult: PredicateResult

): Unit = {

val localCachedUserResult = Option(gizmoduckCache.cache.getIfPresent(candidate.id))

if (localCachedUserResult.isDefined) {

val localPredicateResult = getPredicateResult(request, candidate, localCachedUserResult.get)

localPredicateResult.equals(predicateResult) match {

case true => stats.scope("has\_equal\_predicate\_value").counter("true").incr()

case false => stats.scope("has\_equal\_predicate\_value").counter("false").incr()

}

} else {

stats.scope("has\_equal\_predicate\_value").counter("undefined").incr()

}

}

// method to get PredicateResult from UserResult

def getPredicateResult(

request: HasClientContext with HasParams,

candidate: CandidateUser,

userResult: UserResult,

): PredicateResult = {

userResult.user match {

case Some(user) =>

val abPbReasons = getAbPbReason(user, candidate.getAddressBookMetadata)

val safetyReasons = getSafetyReasons(user)

val countryTakedownReasons = getCountryTakedownReasons(user, request.getCountryCode)

val blinkReasons = getBlinkReasons(user)

val allReasons =

abPbReasons ++ safetyReasons ++ countryTakedownReasons ++ blinkReasons

if (allReasons.nonEmpty) {

invalidPredicateResultCounter.incr()

PredicateResult.Invalid(allReasons)

} else {

validPredicateResultCounter.incr()

PredicateResult.Valid

}

case None =>

noGizmoduckUserCounter.incr()

invalidPredicateResultCounter.incr()

PredicateResult.Invalid(Set(NoUser))

}

}

private def getByUserId(userId: JLong): Stitch[UserResult] = {

StatsUtil.profileStitch(

gizmoduck.getById(userId = userId, queryFields = queryFields, context = lookupContext),

stats.scope("getByUserId")

)

}

}

object GizmoduckPredicate {

private[gizmoduck] val lookupContext: LookupContext =

LookupContext(`includeDeactivated` = true, `safetyLevel` = Some(SafetyLevel.Recommendations))

private[gizmoduck] val queryFields: Set[QueryFields] =

Set(

QueryFields.Discoverability, // needed for Address Book / Phone Book discoverability checks in getAbPbReason

QueryFields.Safety, // needed for user state safety checks in getSafetyReasons, getCountryTakedownReasons

QueryFields.Labels, // needed for user label checks in getBlinkReasons

QueryFields.Takedowns // needed for checking takedown labels for a user in getCountryTakedownReasons

)

private[gizmoduck] val BlinkLabels: Set[LabelValue] = Set(BlinkBad, BlinkWorst)

private[gizmoduck] def getAbPbReason(

user: User,

abMetadataOpt: Option[AddressBookMetadata]

): Set[FilterReason] = {

(for {

discoverability <- user.discoverability

abMetadata <- abMetadataOpt

} yield {

val AddressBookMetadata(fwdPb, rvPb, fwdAb, rvAb) = abMetadata

val abReason: Set[FilterReason] =

if ((!discoverability.discoverableByEmail) && (fwdAb || rvAb))

Set(AddressBookUndiscoverable)

else Set.empty

val pbReason: Set[FilterReason] =

if ((!discoverability.discoverableByMobilePhone) && (fwdPb || rvPb))

Set(PhoneBookUndiscoverable)

else Set.empty

abReason ++ pbReason

}).getOrElse(Set.empty)

}

private[gizmoduck] def getSafetyReasons(user: User): Set[FilterReason] = {

user.safety

.map { s =>

val deactivatedReason: Set[FilterReason] =

if (s.deactivated) Set(Deactivated) else Set.empty

val suspendedReason: Set[FilterReason] = if (s.suspended) Set(Suspended) else Set.empty

val restrictedReason: Set[FilterReason] = if (s.restricted) Set(Restricted) else Set.empty

val nsfwUserReason: Set[FilterReason] = if (s.nsfwUser) Set(NsfwUser) else Set.empty

val nsfwAdminReason: Set[FilterReason] = if (s.nsfwAdmin) Set(NsfwAdmin) else Set.empty

val isProtectedReason: Set[FilterReason] = if (s.isProtected) Set(IsProtected) else Set.empty

deactivatedReason ++ suspendedReason ++ restrictedReason ++ nsfwUserReason ++ nsfwAdminReason ++ isProtectedReason

}.getOrElse(Set.empty)

}

private[gizmoduck] def getCountryTakedownReasons(

user: User,

countryCodeOpt: Option[String]

): Set[FilterReason] = {

(for {

safety <- user.safety.toSeq

if safety.hasTakedown

takedowns <- user.takedowns.toSeq

takedownCountry <- takedowns.countryCodes

requestingCountry <- countryCodeOpt

if takedownCountry.toLowerCase == requestingCountry.toLowerCase

} yield Set(CountryTakedown(takedownCountry.toLowerCase))).flatten.toSet

}

private[gizmoduck] def getBlinkReasons(user: User): Set[FilterReason] = {

user.labels

.map(\_.labels.map(\_.labelValue))

.getOrElse(Nil)

.exists(BlinkLabels.contains)

for {

labels <- user.labels.toSeq

label <- labels.labels

if (BlinkLabels.contains(label.labelValue))

} yield Set(Blink)

}.flatten.toSet

}