package com.twitter.frigate.pushservice.adaptor

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.CandidateSource

import com.twitter.frigate.common.base.CandidateSourceEligible

import com.twitter.frigate.common.base.ListPushCandidate

import com.twitter.frigate.pushservice.model.PushTypes.RawCandidate

import com.twitter.frigate.pushservice.model.PushTypes.Target

import com.twitter.frigate.pushservice.params.PushFeatureSwitchParams

import com.twitter.frigate.pushservice.predicate.TargetPredicates

import com.twitter.frigate.pushservice.util.PushDeviceUtil

import com.twitter.frigate.thriftscala.CommonRecommendationType

import com.twitter.geoduck.service.thriftscala.LocationResponse

import com.twitter.interests\_discovery.thriftscala.DisplayLocation

import com.twitter.interests\_discovery.thriftscala.NonPersonalizedRecommendedLists

import com.twitter.interests\_discovery.thriftscala.RecommendedListsRequest

import com.twitter.interests\_discovery.thriftscala.RecommendedListsResponse

import com.twitter.storehaus.ReadableStore

import com.twitter.util.Future

case class ListsToRecommendCandidateAdaptor(

listRecommendationsStore: ReadableStore[String, NonPersonalizedRecommendedLists],

geoDuckV2Store: ReadableStore[Long, LocationResponse],

idsStore: ReadableStore[RecommendedListsRequest, RecommendedListsResponse],

globalStats: StatsReceiver)

extends CandidateSource[Target, RawCandidate]

with CandidateSourceEligible[Target, RawCandidate] {

override val name: String = this.getClass.getSimpleName

private[this] val stats = globalStats.scope(name)

private[this] val noLocationCodeCounter = stats.counter("no\_location\_code")

private[this] val noCandidatesCounter = stats.counter("no\_candidates\_for\_geo")

private[this] val disablePopGeoListsCounter = stats.counter("disable\_pop\_geo\_lists")

private[this] val disableIDSListsCounter = stats.counter("disable\_ids\_lists")

private def getListCandidate(

targetUser: Target,

\_listId: Long

): RawCandidate with ListPushCandidate = {

new RawCandidate with ListPushCandidate {

override val listId: Long = \_listId

override val commonRecType: CommonRecommendationType = CommonRecommendationType.List

override val target: Target = targetUser

}

}

private def getListsRecommendedFromHistory(

target: Target

): Future[Seq[Long]] = {

target.history.map { history =>

history.sortedHistory.flatMap {

case (\_, notif) if notif.commonRecommendationType == List =>

notif.listNotification.map(\_.listId)

case \_ => None

}

}

}

private def getIDSListRecs(

target: Target,

historicalListIds: Seq[Long]

): Future[Seq[Long]] = {

val request = RecommendedListsRequest(

target.targetId,

DisplayLocation.ListDiscoveryPage,

Some(historicalListIds)

)

if (target.params(PushFeatureSwitchParams.EnableIDSListRecommendations)) {

idsStore.get(request).map {

case Some(response) =>

response.channels.map(\_.id)

case \_ => Nil

}

} else {

disableIDSListsCounter.incr()

Future.Nil

}

}

private def getPopGeoLists(

target: Target,

historicalListIds: Seq[Long]

): Future[Seq[Long]] = {

if (target.params(PushFeatureSwitchParams.EnablePopGeoListRecommendations)) {

geoDuckV2Store.get(target.targetId).flatMap {

case Some(locationResponse) if locationResponse.geohash.isDefined =>

val geoHashLength =

target.params(PushFeatureSwitchParams.ListRecommendationsGeoHashLength)

val geoHash = locationResponse.geohash.get.take(geoHashLength)

listRecommendationsStore

.get(s"geohash\_$geoHash")

.map {

case Some(recommendedLists) =>

recommendedLists.recommendedListsByAlgo.flatMap { topLists =>

topLists.lists.collect {

case list if !historicalListIds.contains(list.listId) => list.listId

}

}

case \_ => Nil

}

case \_ =>

noLocationCodeCounter.incr()

Future.Nil

}

} else {

disablePopGeoListsCounter.incr()

Future.Nil

}

}

override def get(target: Target): Future[Option[Seq[RawCandidate]]] = {

getListsRecommendedFromHistory(target).flatMap { historicalListIds =>

Future

.join(

getPopGeoLists(target, historicalListIds),

getIDSListRecs(target, historicalListIds)

)

.map {

case (popGeoListsIds, idsListIds) =>

val candidates = (idsListIds ++ popGeoListsIds).map(getListCandidate(target, \_))

Some(candidates)

case \_ =>

noCandidatesCounter.incr()

None

}

}

}

private val pushCapFatiguePredicate = TargetPredicates.pushRecTypeFatiguePredicate(

CommonRecommendationType.List,

PushFeatureSwitchParams.ListRecommendationsPushInterval,

PushFeatureSwitchParams.MaxListRecommendationsPushGivenInterval,

stats,

)

override def isCandidateSourceAvailable(target: Target): Future[Boolean] = {

val isNotFatigued = pushCapFatiguePredicate.apply(Seq(target)).map(\_.head)

Future

.join(

PushDeviceUtil.isRecommendationsEligible(target),

isNotFatigued

).map {

case (userRecommendationsEligible, isUnderCAP) =>

userRecommendationsEligible && isUnderCAP && target.params(

PushFeatureSwitchParams.EnableListRecommendations)

}

}

}