package com.twitter.follow\_recommendations.common.candidate\_sources.crowd\_search\_accounts

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

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.follow\_recommendations.common.candidate\_sources.crowd\_search\_accounts.CrowdSearchAccountsParams.AccountsFilteringAndRankingLogics

import com.twitter.follow\_recommendations.common.candidate\_sources.crowd\_search\_accounts.CrowdSearchAccountsParams.CandidateSourceEnabled

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

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

import com.twitter.hermit.model.Algorithm

import com.twitter.onboarding.relevance.crowd\_search\_accounts.thriftscala.CrowdSearchAccounts

import com.twitter.product\_mixer.core.functional\_component.candidate\_source.CandidateSource

import com.twitter.product\_mixer.core.model.common.identifier.CandidateSourceIdentifier

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

import com.twitter.stitch.Stitch

import com.twitter.strato.generated.client.onboarding.userrecs.CrowdSearchAccountsClientColumn

import com.twitter.timelines.configapi.HasParams

import com.twitter.util.Duration

import com.twitter.util.logging.Logging

import javax.inject.Inject

import javax.inject.Singleton

object AccountsFilteringAndRankingLogicId extends Enumeration {

type AccountsFilteringAndRankingLogicId = Value

val NewSearchesDaily: AccountsFilteringAndRankingLogicId = Value("new\_searches\_daily")

val NewSearchesWeekly: AccountsFilteringAndRankingLogicId = Value("new\_searches\_weekly")

val SearchesDaily: AccountsFilteringAndRankingLogicId = Value("searches\_daily")

val SearchesWeekly: AccountsFilteringAndRankingLogicId = Value("searches\_weekly")

}

object CrowdSearchAccountsSource {

val MaxCacheSize = 500

val CacheTTL: Duration = Duration.fromHours(24)

type Target = HasParams with HasClientContext with HasGeohashAndCountryCode

val Identifier: CandidateSourceIdentifier = CandidateSourceIdentifier(

Algorithm.CrowdSearchAccounts.toString)

}

@Singleton

class CrowdSearchAccountsSource @Inject() (

crowdSearchAccountsClientColumn: CrowdSearchAccountsClientColumn,

statsReceiver: StatsReceiver,

) extends CandidateSource[CrowdSearchAccountsSource.Target, CandidateUser]

with Logging {

/\*\* @see [[CandidateSourceIdentifier]] \*/

override val identifier: CandidateSourceIdentifier =

CrowdSearchAccountsSource.Identifier

private val stats = statsReceiver.scope(identifier.name)

private val requestsStats = stats.counter("requests")

private val noCountryCodeStats = stats.counter("no\_country\_code")

private val successStats = stats.counter("success")

private val errorStats = stats.counter("error")

private val cache = StitchCache[String, Option[CrowdSearchAccounts]](

maxCacheSize = CrowdSearchAccountsSource.MaxCacheSize,

ttl = CrowdSearchAccountsSource.CacheTTL,

statsReceiver = statsReceiver.scope(identifier.name, "cache"),

underlyingCall = (k: String) => {

crowdSearchAccountsClientColumn.fetcher

.fetch(k)

.map { result => result.v }

}

)

/\*\* returns a Seq of ''potential'' content \*/

override def apply(

target: CrowdSearchAccountsSource.Target

): Stitch[Seq[CandidateUser]] = {

if (!target.params(CandidateSourceEnabled)) {

return Stitch.value(Seq[CandidateUser]())

}

requestsStats.incr()

target.getCountryCode

.orElse(target.geohashAndCountryCode.flatMap(\_.countryCode)).map { countryCode =>

Stitch

.collect(target

.params(AccountsFilteringAndRankingLogics).map(logic =>

cache.readThrough(countryCode.toUpperCase() + "-" + logic)))

.onSuccess(\_ => {

successStats.incr()

})

.onFailure(t => {

debug("candidate source failed identifier = %s".format(identifier), t)

errorStats.incr()

})

.map(transformCrowdSearchAccountsToCandidateSource)

}.getOrElse {

noCountryCodeStats.incr()

Stitch.value(Seq[CandidateUser]())

}

}

private def transformCrowdSearchAccountsToCandidateSource(

crowdSearchAccounts: Seq[Option[CrowdSearchAccounts]]

): Seq[CandidateUser] = {

crowdSearchAccounts

.flatMap(opt =>

opt

.map(accounts =>

accounts.accounts.map(account =>

CandidateUser(

id = account.accountId,

score = Some(account.searchActivityScore),

).withCandidateSource(identifier)))

.getOrElse(Seq[CandidateUser]()))

}

}