package com.twitter.follow\_recommendations.common.clients.addressbook

import com.twitter.addressbook.datatypes.thriftscala.QueryType

import com.twitter.addressbook.thriftscala.AddressBookGetRequest

import com.twitter.addressbook.thriftscala.AddressBookGetResponse

import com.twitter.addressbook.thriftscala.Addressbook2

import com.twitter.addressbook.thriftscala.ClientInfo

import com.twitter.finagle.stats.NullStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.wtf.scalding.jobs.addressbook.thriftscala.STPResultFeature

import com.twitter.follow\_recommendations.common.clients.addressbook.models.Contact

import com.twitter.follow\_recommendations.common.clients.addressbook.models.EdgeType

import com.twitter.follow\_recommendations.common.clients.addressbook.models.QueryOption

import com.twitter.follow\_recommendations.common.clients.addressbook.models.RecordIdentifier

import com.twitter.wtf.scalding.jobs.address\_book.ABUtil.hashContact

import com.twitter.wtf.scalding.jobs.address\_book.ABUtil.normalizeEmail

import com.twitter.wtf.scalding.jobs.address\_book.ABUtil.normalizePhoneNumber

import com.twitter.hermit.usercontacts.thriftscala.{UserContacts => tUserContacts}

import com.twitter.stitch.Stitch

import com.twitter.strato.client.Fetcher

import javax.inject.Inject

import javax.inject.Singleton

@Singleton

class AddressbookClient @Inject() (

addressbookService: Addressbook2.MethodPerEndpoint,

statsReceiver: StatsReceiver = NullStatsReceiver) {

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

private[this] def getResponseFromService(

identifiers: Seq[RecordIdentifier],

batchSize: Int,

edgeType: EdgeType,

maxFetches: Int,

queryOption: Option[QueryOption]

): Stitch[Seq[AddressBookGetResponse]] = {

Stitch

.collect(

identifiers.map { identifier =>

Stitch.callFuture(

addressbookService.get(AddressBookGetRequest(

clientInfo = ClientInfo(None),

identifier = identifier.toThrift,

edgeType = edgeType.toThrift,

queryType = QueryType.UserId,

queryOption = queryOption.map(\_.toThrift),

maxFetches = maxFetches,

resultBatchSize = batchSize

)))

}

)

}

private[this] def getContactsResponseFromService(

identifiers: Seq[RecordIdentifier],

batchSize: Int,

edgeType: EdgeType,

maxFetches: Int,

queryOption: Option[QueryOption]

): Stitch[Seq[AddressBookGetResponse]] = {

Stitch

.collect(

identifiers.map { identifier =>

Stitch.callFuture(

addressbookService.get(AddressBookGetRequest(

clientInfo = ClientInfo(None),

identifier = identifier.toThrift,

edgeType = edgeType.toThrift,

queryType = QueryType.Contact,

queryOption = queryOption.map(\_.toThrift),

maxFetches = maxFetches,

resultBatchSize = batchSize

)))

}

)

}

/\*\* Mode of addressbook resolving logic

\* ManhattanThenABV2: fetching manhattan cached result and backfill with addressbook v2

\* ABV2Only: calling addressbook v2 directly without fetching manhattan cached result

\* This can be controlled by passing a fetcher or not. Passing a fetcher will attempt to use it,

\* if not then it won't.

\*/

def getUsers(

userId: Long,

identifiers: Seq[RecordIdentifier],

batchSize: Int,

edgeType: EdgeType,

fetcherOption: Option[Fetcher[Long, Unit, tUserContacts]] = None,

maxFetches: Int = 1,

queryOption: Option[QueryOption] = None,

): Stitch[Seq[Long]] = {

fetcherOption match {

case Some(fetcher) =>

getUsersFromManhattan(userId, fetcher).flatMap { userContacts =>

if (userContacts.isEmpty) {

stats.counter("mhEmptyThenFromAbService").incr()

getResponseFromService(identifiers, batchSize, edgeType, maxFetches, queryOption)

.map(\_.flatMap(\_.users).flatten.distinct)

} else {

stats.counter("fromManhattan").incr()

Stitch.value(userContacts)

}

}

case None =>

stats.counter("fromAbService").incr()

getResponseFromService(identifiers, batchSize, edgeType, maxFetches, queryOption)

.map(\_.flatMap(\_.users).flatten.distinct)

}

}

def getHashedContacts(

normalizeFn: String => String,

extractField: String,

)(

userId: Long,

identifiers: Seq[RecordIdentifier],

batchSize: Int,

edgeType: EdgeType,

fetcherOption: Option[Fetcher[String, Unit, STPResultFeature]] = None,

maxFetches: Int = 1,

queryOption: Option[QueryOption] = None,

): Stitch[Seq[String]] = {

fetcherOption match {

case Some(fetcher) =>

getContactsFromManhattan(userId, fetcher).flatMap { userContacts =>

if (userContacts.isEmpty) {

getContactsResponseFromService(

identifiers,

batchSize,

edgeType,

maxFetches,

queryOption)

.map { response =>

for {

resp <- response

contacts <- resp.contacts

contactsThrift = contacts.map(Contact.fromThrift)

contactsSet = extractField match {

case "emails" => contactsThrift.flatMap(\_.emails.toSeq.flatten)

case "phoneNumbers" => contactsThrift.flatMap(\_.phoneNumbers.toSeq.flatten)

}

hashedAndNormalizedContacts = contactsSet.map(c => hashContact(normalizeFn(c)))

} yield hashedAndNormalizedContacts

}.map(\_.flatten)

} else {

Stitch.Nil

}

}

case None => {

getContactsResponseFromService(identifiers, batchSize, edgeType, maxFetches, queryOption)

.map { response =>

for {

resp <- response

contacts <- resp.contacts

contactsThrift = contacts.map(Contact.fromThrift)

contactsSet = extractField match {

case "emails" => contactsThrift.flatMap(\_.emails.toSeq.flatten)

case "phoneNumbers" => contactsThrift.flatMap(\_.phoneNumbers.toSeq.flatten)

}

hashedAndNormalizedContacts = contactsSet.map(c => hashContact(normalizeFn(c)))

} yield hashedAndNormalizedContacts

}.map(\_.flatten)

}

}

}

def getEmailContacts = getHashedContacts(normalizeEmail, "emails") \_

def getPhoneContacts = getHashedContacts(normalizePhoneNumber, "phoneNumbers") \_

private def getUsersFromManhattan(

userId: Long,

fetcher: Fetcher[Long, Unit, tUserContacts],

): Stitch[Seq[Long]] = fetcher

.fetch(userId)

.map(\_.v.map(\_.destinationIds).toSeq.flatten.distinct)

private def getContactsFromManhattan(

userId: Long,

fetcher: Fetcher[String, Unit, STPResultFeature],

): Stitch[Seq[String]] = fetcher

.fetch(userId.toString)

.map(\_.v.map(\_.strongTieUserFeature.map(\_.destId)).toSeq.flatten.distinct)

}

object AddressbookClient {

val AddressBook2BatchSize = 500

def createQueryOption(edgeType: EdgeType, isPhone: Boolean): Option[QueryOption] =

(edgeType, isPhone) match {

case (EdgeType.Reverse, \_) =>

None

case (EdgeType.Forward, true) =>

Some(

QueryOption(

onlyDiscoverableInExpansion = false,

onlyConfirmedInExpansion = false,

onlyDiscoverableInResult = false,

onlyConfirmedInResult = false,

fetchGlobalApiNamespace = false,

isDebugRequest = false,

resolveEmails = false,

resolvePhoneNumbers = true

))

case (EdgeType.Forward, false) =>

Some(

QueryOption(

onlyDiscoverableInExpansion = false,

onlyConfirmedInExpansion = false,

onlyDiscoverableInResult = false,

onlyConfirmedInResult = false,

fetchGlobalApiNamespace = false,

isDebugRequest = false,

resolveEmails = true,

resolvePhoneNumbers = false

))

}

}