package com.twitter.frigate.pushservice.take

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.logger.MRLogger

import com.twitter.frigate.common.ntab.InvalidNTABWriteRequestException

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

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

import com.twitter.frigate.pushservice.params.PushParams

import com.twitter.frigate.thriftscala.CommonRecommendationType

import com.twitter.gizmoduck.thriftscala.User

import com.twitter.notificationservice.thriftscala.\_

import com.twitter.storehaus.ReadableStore

import com.twitter.timelines.configapi.Param

import com.twitter.util.Future

import scala.util.control.NoStackTrace

class NtabCopyIdNotFoundException(private val message: String)

extends Exception(message)

with NoStackTrace

class InvalidNtabCopyIdException(private val message: String)

extends Exception(message)

with NoStackTrace

object NotificationServiceSender {

def generateSocialContextTextEntities(

ntabDisplayNamesAndIdsFut: Future[Seq[(String, Long)]],

otherCountFut: Future[Int]

): Future[Seq[DisplayTextEntity]] = {

Future.join(ntabDisplayNamesAndIdsFut, otherCountFut).map {

case (namesWithIdInOrder, otherCount) =>

val displays = namesWithIdInOrder.zipWithIndex.map {

case ((name, id), index) =>

DisplayTextEntity(

name = "user" + s"${index + 1}",

value = TextValue.Text(name),

emphasis = true,

userId = Some(id)

)

} ++ Seq(

DisplayTextEntity(name = "nameCount", value = TextValue.Number(namesWithIdInOrder.size))

)

val otherDisplay = if (otherCount > 0) {

Some(

DisplayTextEntity(

name = "otherCount",

value = TextValue.Number(otherCount)

)

)

} else None

displays ++ otherDisplay

}

}

def getDisplayTextEntityFromUser(

userOpt: Option[User],

fieldName: String,

isBold: Boolean

): Option[DisplayTextEntity] = {

for {

user <- userOpt

profile <- user.profile

} yield {

DisplayTextEntity(

name = fieldName,

value = TextValue.Text(profile.name),

emphasis = isBold,

userId = Some(user.id)

)

}

}

def getDisplayTextEntityFromUser(

user: Future[Option[User]],

fieldName: String,

isBold: Boolean

): Future[Option[DisplayTextEntity]] = {

user.map { getDisplayTextEntityFromUser(\_, fieldName, isBold) }

}

}

case class NotificationServiceRequest(

candidate: PushCandidate,

impressionId: String,

isBadgeUpdate: Boolean,

overrideId: Option[String] = None)

class NotificationServiceSender(

send: (Target, CreateGenericNotificationRequest) => Future[CreateGenericNotificationResponse],

enableWritesParam: Param[Boolean],

enableForEmployeesParam: Param[Boolean],

enableForEveryoneParam: Param[Boolean]

)(

implicit globalStats: StatsReceiver)

extends ReadableStore[NotificationServiceRequest, CreateGenericNotificationResponse] {

val log = MRLogger(this.getClass.getName)

val stats = globalStats.scope("NotificationServiceSender")

val requestEmpty = stats.scope("request\_empty")

val requestNonEmpty = stats.counter("request\_non\_empty")

val requestBadgeCount = stats.counter("request\_badge\_count")

val successfulWrite = stats.counter("successful\_write")

val successfulWriteScope = stats.scope("successful\_write")

val failedWriteScope = stats.scope("failed\_write")

val gotNonSuccessResponse = stats.counter("got\_non\_success\_response")

val gotEmptyResponse = stats.counter("got\_empty\_response")

val deciderTurnedOffResponse = stats.scope("decider\_turned\_off\_response")

val disabledByDeciderForCandidate = stats.scope("model/candidate").counter("disabled\_by\_decider")

val sentToAlphaUserForCandidate =

stats.scope("model/candidate").counter("send\_to\_employee\_or\_team")

val sentToNonBucketedUserForCandidate =

stats.scope("model/candidate").counter("send\_to\_non\_bucketed\_decidered\_user")

val noSendForCandidate = stats.scope("model/candidate").counter("no\_send")

val ineligibleUsersForCandidate = stats.scope("model/candidate").counter("ineligible\_users")

val darkWriteRequestsForCandidate = stats.scope("model/candidate").counter("dark\_write\_traffic")

val heavyUserForCandidateCounter = stats.scope("model/candidate").counter("target\_heavy")

val nonHeavyUserForCandidateCounter = stats.scope("model/candidate").counter("target\_non\_heavy")

val skipWritingToNTAB = stats.counter("skip\_writing\_to\_ntab")

val ntabWriteDisabledForCandidate = stats.scope("model/candidate").counter("ntab\_write\_disabled")

val ntabOverrideEnabledForCandidate = stats.scope("model/candidate").counter("override\_enabled")

val ntabTTLForCandidate = stats.scope("model/candidate").counter("ttl\_enabled")

override def get(

notifRequest: NotificationServiceRequest

): Future[Option[CreateGenericNotificationResponse]] = {

notifRequest.candidate.target.deviceInfo.flatMap { deviceInfoOpt =>

val disableWritingToNtab =

notifRequest.candidate.target.params(PushParams.DisableWritingToNTAB)

if (disableWritingToNtab) {

skipWritingToNTAB.incr()

Future.None

} else {

if (notifRequest.overrideId.nonEmpty) { ntabOverrideEnabledForCandidate.incr() }

Future

.join(

notifRequest.candidate.ntabRequest,

ntabWritesEnabledForCandidate(notifRequest.candidate)).flatMap {

case (Some(ntabRequest), ntabWritesEnabled) if ntabWritesEnabled =>

if (ntabRequest.expiryTimeMillis.nonEmpty) { ntabTTLForCandidate.incr() }

sendNTabRequest(

ntabRequest,

notifRequest.candidate.target,

notifRequest.isBadgeUpdate,

notifRequest.candidate.commonRecType,

isFromCandidate = true,

overrideId = notifRequest.overrideId

)

case (Some(\_), ntabWritesEnabled) if !ntabWritesEnabled =>

ntabWriteDisabledForCandidate.incr()

Future.None

case (None, ntabWritesEnabled) =>

if (!ntabWritesEnabled) ntabWriteDisabledForCandidate.incr()

requestEmpty.counter(s"candidate\_${notifRequest.candidate.commonRecType}").incr()

Future.None

}

}

}

}

private def sendNTabRequest(

genericNotificationRequest: CreateGenericNotificationRequest,

target: Target,

isBadgeUpdate: Boolean,

crt: CommonRecommendationType,

isFromCandidate: Boolean,

overrideId: Option[String]

): Future[Option[CreateGenericNotificationResponse]] = {

requestNonEmpty.incr()

val notifSvcReq =

genericNotificationRequest.copy(

sendBadgeCountUpdate = isBadgeUpdate,

overrideId = overrideId

)

requestBadgeCount.incr()

send(target, notifSvcReq)

.map { response =>

if (response.responseType.equals(CreateGenericNotificationResponseType.DecideredOff)) {

deciderTurnedOffResponse.counter(s"$crt").incr()

deciderTurnedOffResponse.counter(s"${genericNotificationRequest.genericType}").incr()

throw InvalidNTABWriteRequestException("Decider is turned off")

} else {

Some(response)

}

}

.onFailure { ex =>

stats.counter(s"error\_${ex.getClass.getCanonicalName}").incr()

failedWriteScope.counter(s"${crt}").incr()

log

.error(

ex,

s"NTAB failure $notifSvcReq"

)

}

.onSuccess {

case Some(response) =>

successfulWrite.incr()

val successfulWriteScopeString = if (isFromCandidate) "model/candidate" else "envelope"

successfulWriteScope.scope(successfulWriteScopeString).counter(s"$crt").incr()

if (response.responseType != CreateGenericNotificationResponseType.Success) {

gotNonSuccessResponse.incr()

log.warning(s"NTAB dropped $notifSvcReq with response $response")

}

case \_ =>

gotEmptyResponse.incr()

}

}

private def ntabWritesEnabledForCandidate(cand: PushCandidate): Future[Boolean] = {

if (!cand.target.params(enableWritesParam)) {

disabledByDeciderForCandidate.incr()

Future.False

} else {

Future

.join(

cand.target.isAnEmployee,

cand.target.isInNotificationsServiceWhitelist,

cand.target.isTeamMember

)

.flatMap {

case (isEmployee, isInNotificationsServiceWhitelist, isTeamMember) =>

cand.target.deviceInfo.flatMap { deviceInfoOpt =>

deviceInfoOpt

.map { deviceInfo =>

cand.target.isHeavyUserState.map { isHeavyUser =>

val isAlphaTester = (isEmployee && cand.target

.params(enableForEmployeesParam)) || isInNotificationsServiceWhitelist || isTeamMember

if (cand.target.isDarkWrite) {

stats

.scope("model/candidate").counter(

s"dark\_write\_${cand.commonRecType}").incr()

darkWriteRequestsForCandidate.incr()

false

} else if (isAlphaTester || deviceInfo.isMRinNTabEligible

|| cand.target.insertMagicrecsIntoNTabForNonPushableUsers) {

if (isHeavyUser) heavyUserForCandidateCounter.incr()

else nonHeavyUserForCandidateCounter.incr()

val enabledForDesiredUsers = cand.target.params(enableForEveryoneParam)

if (isAlphaTester) {

sentToAlphaUserForCandidate.incr()

true

} else if (enabledForDesiredUsers) {

sentToNonBucketedUserForCandidate.incr()

true

} else {

noSendForCandidate.incr()

false

}

} else {

ineligibleUsersForCandidate.incr()

false

}

}

}.getOrElse(Future.False)

}

}

}

}

}