package com.twitter.frigate.pushservice.take.sender

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.TweetCandidate

import com.twitter.frigate.common.base.TweetDetails

import com.twitter.frigate.common.store.IbisResponse

import com.twitter.frigate.common.store.InvalidConfiguration

import com.twitter.frigate.common.store.NoRequest

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

import com.twitter.frigate.pushservice.params.{PushFeatureSwitchParams => FS}

import com.twitter.frigate.pushservice.store.Ibis2Store

import com.twitter.frigate.pushservice.store.TweetTranslationStore

import com.twitter.frigate.pushservice.util.CopyUtil

import com.twitter.frigate.pushservice.util.FunctionalUtil

import com.twitter.frigate.pushservice.util.InlineActionUtil

import com.twitter.frigate.pushservice.util.OverrideNotificationUtil

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

import com.twitter.frigate.scribe.thriftscala.NotificationScribe

import com.twitter.frigate.thriftscala.ChannelName

import com.twitter.frigate.thriftscala.NotificationDisplayLocation

import com.twitter.ibis2.service.thriftscala.Ibis2Request

import com.twitter.notificationservice.thriftscala.CreateGenericNotificationResponse

import com.twitter.storehaus.ReadableStore

import com.twitter.util.Future

class Ibis2Sender(

pushIbisV2Store: Ibis2Store,

tweetTranslationStore: ReadableStore[TweetTranslationStore.Key, TweetTranslationStore.Value],

statsReceiver: StatsReceiver) {

private val stats = statsReceiver.scope(getClass.getSimpleName)

private val silentPushCounter = stats.counter("silent\_push")

private val ibisSendFailureCounter = stats.scope("ibis\_send\_failure").counter("failures")

private val buggyAndroidReleaseCounter = stats.counter("is\_buggy\_android\_release")

private val androidPrimaryCounter = stats.counter("android\_primary\_device")

private val addTranslationModelValuesCounter = stats.counter("with\_translation\_model\_values")

private val patchNtabResponseEnabled = stats.scope("with\_ntab\_response")

private val noIbisPushStats = stats.counter("no\_ibis\_push")

private def ibisSend(

candidate: PushCandidate,

translationModelValues: Option[Map[String, String]] = None,

ntabResponse: Option[CreateGenericNotificationResponse] = None

): Future[IbisResponse] = {

if (candidate.frigateNotification.notificationDisplayLocation != NotificationDisplayLocation.PushToMobileDevice) {

Future.value(IbisResponse(InvalidConfiguration))

} else {

candidate.ibis2Request.flatMap {

case Some(request) =>

val requestWithTranslationMV =

addTranslationModelValues(request, translationModelValues)

val patchedIbisRequest = {

if (candidate.target.isLoggedOutUser) {

requestWithTranslationMV

} else {

patchNtabResponseToIbisRequest(requestWithTranslationMV, candidate, ntabResponse)

}

}

pushIbisV2Store.send(patchedIbisRequest, candidate)

case \_ =>

noIbisPushStats.incr()

Future.value(IbisResponse(sendStatus = NoRequest, ibis2Response = None))

}

}

}

def sendAsDarkWrite(

candidate: PushCandidate

): Future[IbisResponse] = {

ibisSend(candidate)

}

def send(

channels: Seq[ChannelName],

pushCandidate: PushCandidate,

notificationScribe: NotificationScribe => Unit,

ntabResponse: Option[CreateGenericNotificationResponse],

): Future[IbisResponse] = pushCandidate.target.isSilentPush.flatMap { isSilentPush: Boolean =>

if (isSilentPush) silentPushCounter.incr()

pushCandidate.target.deviceInfo.flatMap { deviceInfo =>

if (deviceInfo.exists(\_.isSim40AndroidVersion)) buggyAndroidReleaseCounter.incr()

if (PushDeviceUtil.isPrimaryDeviceAndroid(deviceInfo)) androidPrimaryCounter.incr()

Future

.join(

OverrideNotificationUtil

.getOverrideInfo(pushCandidate, stats),

CopyUtil.getCopyFeatures(pushCandidate, stats),

getTranslationModelValues(pushCandidate)

).flatMap {

case (overrideInfoOpt, copyFeaturesMap, translationModelValues) =>

ibisSend(pushCandidate, translationModelValues, ntabResponse)

.onSuccess { ibisResponse =>

pushCandidate

.scribeData(

ibis2Response = ibisResponse.ibis2Response,

isSilentPush = isSilentPush,

overrideInfoOpt = overrideInfoOpt,

copyFeaturesList = copyFeaturesMap.keySet,

channels = channels

).foreach(notificationScribe)

}.onFailure { \_ =>

pushCandidate

.scribeData(channels = channels).foreach { data =>

ibisSendFailureCounter.incr()

notificationScribe(data)

}

}

}

}

}

private def getTranslationModelValues(

candidate: PushCandidate

): Future[Option[Map[String, String]]] = {

candidate match {

case tweetCandidate: TweetCandidate with TweetDetails =>

val key = TweetTranslationStore.Key(

target = candidate.target,

tweetId = tweetCandidate.tweetId,

tweet = tweetCandidate.tweet,

crt = candidate.commonRecType

)

tweetTranslationStore

.get(key)

.map {

case Some(value) =>

Some(

Map(

"translated\_tweet\_text" -> value.translatedTweetText,

"localized\_source\_language" -> value.localizedSourceLanguage

))

case None => None

}

case \_ => Future.None

}

}

private def addTranslationModelValues(

ibisRequest: Ibis2Request,

translationModelValues: Option[Map[String, String]]

): Ibis2Request = {

(translationModelValues, ibisRequest.modelValues) match {

case (Some(translationModelVal), Some(existingModelValues)) =>

addTranslationModelValuesCounter.incr()

ibisRequest.copy(modelValues = Some(translationModelVal ++ existingModelValues))

case (Some(translationModelVal), None) =>

addTranslationModelValuesCounter.incr()

ibisRequest.copy(modelValues = Some(translationModelVal))

case (None, \_) => ibisRequest

}

}

private def patchNtabResponseToIbisRequest(

ibis2Req: Ibis2Request,

candidate: PushCandidate,

ntabResponse: Option[CreateGenericNotificationResponse]

): Ibis2Request = {

if (candidate.target.params(FS.EnableInlineFeedbackOnPush)) {

patchNtabResponseEnabled.counter().incr()

val dislikePosition = candidate.target.params(FS.InlineFeedbackSubstitutePosition)

val dislikeActionOption = ntabResponse

.map(FunctionalUtil.incr(patchNtabResponseEnabled.counter("ntab\_response\_exist")))

.flatMap(response => InlineActionUtil.getDislikeInlineAction(candidate, response))

.map(FunctionalUtil.incr(patchNtabResponseEnabled.counter("dislike\_action\_generated")))

// Only generate patch serialized inline action when original request has existing serialized\_inline\_actions\_v2

val patchedSerializedActionOption = ibis2Req.modelValues

.flatMap(model => model.get("serialized\_inline\_actions\_v2"))

.map(FunctionalUtil.incr(patchNtabResponseEnabled.counter("inline\_action\_v2\_exists")))

.map(serialized =>

InlineActionUtil

.patchInlineActionAtPosition(serialized, dislikeActionOption, dislikePosition))

.map(FunctionalUtil.incr(patchNtabResponseEnabled.counter("patch\_inline\_action\_generated")))

(ibis2Req.modelValues, patchedSerializedActionOption) match {

case (Some(existingModelValue), Some(patchedActionV2)) =>

patchNtabResponseEnabled.scope("patch\_applied").counter().incr()

ibis2Req.copy(modelValues =

Some(existingModelValue ++ Map("serialized\_inline\_actions\_v2" -> patchedActionV2)))

case \_ => ibis2Req

}

} else ibis2Req

}

}