package com.twitter.tweetypie

package handler

import com.twitter.expandodo.thriftscala.Card2RequestOptions

import com.twitter.featureswitches.v2.FeatureSwitchResults

import com.twitter.gizmoduck.util.UserUtil

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.core.TweetCreateFailure

import com.twitter.tweetypie.repository.Card2Repository

import com.twitter.tweetypie.repository.StratoPromotedTweetRepository

import com.twitter.tweetypie.repository.StratoSubscriptionVerificationRepository

import com.twitter.tweetypie.repository.TweetQuery

import com.twitter.tweetypie.repository.TweetRepository

import com.twitter.tweetypie.repository.UrlCard2Key

import com.twitter.tweetypie.thriftscala.EditControl

import com.twitter.tweetypie.thriftscala.EditOptions

import com.twitter.tweetypie.thriftscala.TweetCreateState

import com.twitter.tweetypie.util.EditControlUtil.\_

import com.twitter.tweetypie.thriftscala.CardReference

import com.twitter.tweetypie.thriftscala.EditControlInitial

import com.twitter.tweetypie.thriftscala.PostTweetRequest

import com.twitter.tweetypie.util.CommunityAnnotation

import com.twitter.tweetypie.util.EditControlUtil

import com.twitter.util.Future

object EditControlBuilder {

type Type = Request => Future[Option[EditControl]]

val editTweetCountStat = "edit\_tweet\_count"

val editControlQueryOptions = TweetQuery.Options(

TweetQuery.Include(Set(Tweet.CoreDataField.id, Tweet.EditControlField.id))

)

val TweetEditCreationEnabledKey = "tweet\_edit\_creation\_enabled"

val TweetEditCreationEnabledForTwitterBlueKey = "tweet\_edit\_creation\_enabled\_for\_twitter\_blue"

val pollCardNames: Set[String] = Set(

"poll2choice\_text\_only",

"poll3choice\_text\_only",

"poll4choice\_text\_only",

"poll2choice\_image",

"poll3choice\_image",

"poll4choice\_image",

"poll2choice\_video",

"poll3choice\_video",

"poll4choice\_video",

)

/\*\* Used just for checking card name for poll check in case cards platform key not provided. \*/

val defaultCardsPlatformKey = "iPhone-13"

/\*\*

\* Do we assume a Tweet has a poll (which makes it not editable) when it has a card

\* that could be a poll, and it cannot be resolved at create.

\*/

val isPollCardAssumption = true

val tweetEditSubscriptionResource = "feature/tweet\_edit"

val log: Logger = Logger(getClass)

case class Request(

postTweetRequest: PostTweetRequest,

tweet: Tweet,

matchedResults: Option[FeatureSwitchResults]) {

def editOptions: Option[EditOptions] = postTweetRequest.editOptions

def authorId: UserId = postTweetRequest.userId

def createdAt: Time = Time.fromMilliseconds(tweet.coreData.get.createdAtSecs \* 1000L)

def tweetId: TweetId = tweet.id

def cardReference: Option[CardReference] =

postTweetRequest.additionalFields.flatMap(\_.cardReference)

def cardsPlatformKey: Option[String] =

postTweetRequest.hydrationOptions.flatMap(\_.cardsPlatformKey)

}

def apply(

tweetRepo: TweetRepository.Type,

card2Repo: Card2Repository.Type,

promotedTweetRepo: StratoPromotedTweetRepository.Type,

subscriptionVerificationRepo: StratoSubscriptionVerificationRepository.Type,

disablePromotedTweetEdit: Gate[Unit],

checkTwitterBlueSubscription: Gate[Unit],

setEditWindowToSixtyMinutes: Gate[Unit],

stats: StatsReceiver

): Type = {

// Nullcast tweets not allowed, except if the tweet has a community annotation

def isNullcastedButNotCommunityTweet(request: Request): Boolean = {

val isNullcasted: Boolean = request.tweet.coreData.get.nullcast

val communityIds: Option[Seq[CommunityId]] =

request.postTweetRequest.additionalFields

.flatMap(CommunityAnnotation.additionalFieldsToCommunityIDs)

isNullcasted && !(communityIds.exists(\_.nonEmpty))

}

def isSuperFollow(tweet: Tweet): Boolean = tweet.exclusiveTweetControl.isDefined

def isCollabTweet(tweet: Tweet): Boolean = tweet.collabControl.isDefined

def isReplyToTweet(tweet: Tweet): Boolean =

getReply(tweet).flatMap(\_.inReplyToStatusId).isDefined

// When card is tombstone, tweet is not considered a poll, and therefore can be edit eligible.

val cardReferenceUriIsTombstone = stats.counter("edit\_control\_builder\_card\_tombstoned")

// We check whether tweets are polls since these are not edit eligible.

// If we are not sure due to lookup failure, we take an `isPollCardAssumption`.

def isPoll(

card2Repo: Card2Repository.Type,

cardReference: CardReference,

cardsPlatformKey: String,

): Stitch[Boolean] = {

if (cardReference.cardUri == "tombstone://card") {

cardReferenceUriIsTombstone.incr()

Stitch.value(false)

} else {

val key = UrlCard2Key(cardReference.cardUri)

// `allowNonTcoUrls = true` This allows us to check if non-tco urls (e.g. apple.com) have a card

// at this point in tweet builder urls can be in their original form and not tcoified.

val options = Card2RequestOptions(

platformKey = cardsPlatformKey,

allowNonTcoUrls = true

)

card2Repo(key, options)

.map(card2 => pollCardNames.contains(card2.name))

}

}

def isFeatureSwitchEnabled(matchedResults: Option[FeatureSwitchResults], key: String): Boolean =

matchedResults.flatMap(\_.getBoolean(key, shouldLogImpression = false)).contains(true)

def wrapInitial(initial: EditControlInitial): Option[EditControl.Initial] =

Some(EditControl.Initial(initial = initial))

// Checks for validity of an edit are implemented as procedures

// that throw an error in case a check fails. This composes way better than

// returning a Try/Future/Stitch because:

// 1. We do not need to decide which of the aforementioned containers to use.

// 2. The checks as below compose with callbacks in all the aforementioned containers.

val editRequestOutsideOfAllowlist = stats.counter("edit\_control\_builder\_rejected", "allowlist")

// This method uses two feature switches:

// - TweetEditCreationEnabledKey authorizes the user to edit tweets directly

// - TweetEditCreationEnabledForTwitterBlueKey authorizes the user to edit tweets if they have

// a Twitter Blue subscription

//

// Test users are always authorized to edit tweets.

def checkUserEligibility(

authorId: UserId,

matchedResults: Option[FeatureSwitchResults]

): Stitch[Unit] = {

val isTestUser = UserUtil.isTestUserId(authorId)

val authorizedWithoutTwitterBlue =

isFeatureSwitchEnabled(matchedResults, TweetEditCreationEnabledKey)

if (isTestUser || authorizedWithoutTwitterBlue) {

// If the editing user is a test user or is authorized by the non-Twitter Blue feature

// switch, allow editing.

Stitch.Done

} else {

// Otherwise, check if they're authorized by the Twitter Blue feature switch and if they're

// subscribed to Twitter Blue.

val authorizedWithTwitterBlue: Stitch[Boolean] =

if (checkTwitterBlueSubscription() &&

isFeatureSwitchEnabled(matchedResults, TweetEditCreationEnabledForTwitterBlueKey)) {

subscriptionVerificationRepo(authorId, tweetEditSubscriptionResource)

} else Stitch.value(false)

authorizedWithTwitterBlue.flatMap { authorized =>

if (!authorized) {

log.error(s"User ${authorId} unauthorized to edit")

editRequestOutsideOfAllowlist.incr()

Stitch.exception(TweetCreateFailure.State(TweetCreateState.EditTweetUserNotAuthorized))

} else Stitch.Done

}

}

}

val editRequestByNonAuthor = stats.counter("edit\_control\_builder\_rejected", "not\_author")

def checkAuthor(

authorId: UserId,

previousTweetAuthorId: UserId

): Unit = {

if (authorId != previousTweetAuthorId) {

editRequestByNonAuthor.incr()

throw TweetCreateFailure.State(TweetCreateState.EditTweetUserNotAuthor)

}

}

val tweetEditForStaleTweet = stats.counter("edit\_control\_builder\_rejected", "stale")

def checkLatestEdit(

previousTweetId: TweetId,

initial: EditControlInitial,

): Unit = {

if (previousTweetId != initial.editTweetIds.last) {

tweetEditForStaleTweet.incr()

throw TweetCreateFailure.State(TweetCreateState.EditTweetNotLatestVersion)

}

}

val tweetEditForLimitReached = stats.counter("edit\_control\_builder\_rejected", "edits\_limit")

def checkEditsRemaining(initial: EditControlInitial): Unit = {

initial.editsRemaining match {

case Some(number) if number > 0 => // OK

case \_ =>

tweetEditForLimitReached.incr()

throw TweetCreateFailure.State(TweetCreateState.EditCountLimitReached)

}

}

val editTweetExpired = stats.counter("edit\_control\_builder\_rejected", "expired")

val editTweetExpiredNoEditControl =

stats.counter("edit\_control\_builder\_rejected", "expired", "no\_edit\_control")

def checkEditTimeWindow(initial: EditControlInitial): Unit = {

initial.editableUntilMsecs match {

case Some(millis) if Time.now < Time.fromMilliseconds(millis) => // OK

case Some(\_) =>

editTweetExpired.incr()

throw TweetCreateFailure.State(TweetCreateState.EditTimeLimitReached)

case editable =>

editTweetExpired.incr()

if (editable.isEmpty) {

editTweetExpiredNoEditControl.incr()

}

throw TweetCreateFailure.State(TweetCreateState.EditTimeLimitReached)

}

}

val tweetEditNotEligible = stats.counter("edit\_control\_builder\_rejected", "not\_eligible")

def checkIsEditEligible(initial: EditControlInitial): Unit = {

initial.isEditEligible match {

case Some(true) => // OK

case \_ =>

tweetEditNotEligible.incr()

throw TweetCreateFailure.State(TweetCreateState.NotEligibleForEdit)

}

}

val editControlInitialMissing =

stats.counter("edit\_control\_builder\_rejected", "initial\_missing")

def findEditControlInitial(previousTweet: Tweet): EditControlInitial = {

previousTweet.editControl match {

case Some(EditControl.Initial(initial)) => initial

case Some(EditControl.Edit(edit)) =>

edit.editControlInitial.getOrElse {

editControlInitialMissing.incr()

throw new IllegalStateException(

"Encountered edit tweet with missing editControlInitial.")

}

case \_ =>

throw TweetCreateFailure.State(TweetCreateState.EditTimeLimitReached)

}

}

val editPromotedTweet = stats.counter("tweet\_edit\_for\_promoted\_tweet")

def checkPromotedTweet(

previousTweetId: TweetId,

promotedTweetRepo: StratoPromotedTweetRepository.Type,

disablePromotedTweetEdit: Gate[Unit]

): Stitch[Unit] = {

if (disablePromotedTweetEdit()) {

promotedTweetRepo(previousTweetId).flatMap {

case false =>

Stitch.Done

case true =>

editPromotedTweet.incr()

Stitch.exception(TweetCreateFailure.State(TweetCreateState.EditTweetUserNotAuthorized))

}

} else {

Stitch.Done

}

}

// Each time edit is made, count how many versions a tweet already has.

// Value should be always between 1 and 4.

val editTweetCount = 0

.to(EditControlUtil.maxTweetEditsAllowed)

.map(i => i -> stats.counter("edit\_control\_builder\_edits\_count", i.toString))

.toMap

// Overall counter and failures of card resolution for poll lookups. Needed because polls are not editable.

val pollCardResolutionTotal = stats.counter("edit\_control\_builder\_card\_resolution", "total")

val pollCardResolutionFailure =

stats.counter("edit\_control\_builder\_card\_resolution", "failures")

// Edit of initial tweet requested, and all edit checks successful.

val initialEditTweet = stats.counter("edit\_control\_builder\_initial\_edit")

request =>

Stitch.run {

request.editOptions match {

case None =>

val editControl =

makeEditControlInitial(

tweetId = request.tweetId,

createdAt = request.createdAt,

setEditWindowToSixtyMinutes = setEditWindowToSixtyMinutes

).initial.copy(

isEditEligible = Some(

!isNullcastedButNotCommunityTweet(request)

&& !isSuperFollow(request.tweet)

&& !isCollabTweet(request.tweet)

&& !isReplyToTweet(request.tweet)

),

)

(editControl.isEditEligible, request.cardReference) match {

case (Some(true), Some(reference)) =>

pollCardResolutionTotal.incr()

isPoll(

card2Repo = card2Repo,

cardReference = reference,

cardsPlatformKey = request.cardsPlatformKey.getOrElse(defaultCardsPlatformKey),

).rescue {

// Revert to the assumed value if card cannot be resolved.

case \_ =>

pollCardResolutionFailure.incr()

Stitch.value(isPollCardAssumption)

}

.map { tweetIsAPoll =>

wrapInitial(editControl.copy(isEditEligible = Some(!tweetIsAPoll)))

}

case \_ => Stitch.value(wrapInitial(editControl))

}

case Some(editOptions) =>

for {

(previousTweet, \_, \_) <- Stitch.join(

tweetRepo(editOptions.previousTweetId, editControlQueryOptions),

checkPromotedTweet(

editOptions.previousTweetId,

promotedTweetRepo,

disablePromotedTweetEdit),

checkUserEligibility(

authorId = request.authorId,

matchedResults = request.matchedResults)

)

} yield {

val initial = findEditControlInitial(previousTweet)

checkAuthor(

authorId = request.authorId,

previousTweetAuthorId = getUserId(previousTweet))

editTweetCount

.get(initial.editTweetIds.size)

.orElse(editTweetCount.get(EditControlUtil.maxTweetEditsAllowed))

.foreach(counter => counter.incr())

checkLatestEdit(previousTweet.id, initial)

checkEditsRemaining(initial)

checkEditTimeWindow(initial)

checkIsEditEligible(initial)

if (initial.editTweetIds == Seq(previousTweet.id)) {

initialEditTweet.incr()

}

Some(editControlEdit(initialTweetId = initial.editTweetIds.head))

}

}

}

}

}