package com.twitter.tweetypie.handler

import com.twitter.featureswitches.v2.FeatureSwitchResults

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.UserId

import com.twitter.tweetypie.\_

import com.twitter.tweetypie.core.TweetCreateFailure

import com.twitter.tweetypie.repository.UserIdentityRepository

import com.twitter.tweetypie.repository.UserKey

import com.twitter.tweetypie.thriftscala.ConversationControl

import com.twitter.tweetypie.thriftscala.Tweet

import com.twitter.tweetypie.thriftscala.TweetCreateConversationControl

import com.twitter.tweetypie.thriftscala.TweetCreateState.ConversationControlNotAllowed

import com.twitter.tweetypie.thriftscala.TweetCreateState.InvalidConversationControl

import com.twitter.tweetypie.util.ConversationControls

import com.twitter.util.logging.Logging

/\*\*

\* Process request parameters into a ConversationControl value.

\*/

object ConversationControlBuilder extends Logging {

type Type = Request => Stitch[Option[ConversationControl]]

type ScreenName = String

/\*\*

\* The fields necessary to create a [[ConversationControl]].

\*

\* This is a trait rather than a case class to avoid running the

\* code to extract the mentions in the cases where handling the

\* request doesn't need to use them (the common case where

\* tweetCreateConversationControl is None).

\*/

trait Request {

def tweetCreateConversationControl: Option[TweetCreateConversationControl]

def tweetAuthorId: UserId

def mentionedUserScreenNames: Set[String]

def noteTweetMentionedUserIds: Option[Set[Long]]

}

object Request {

/\*\*

\* Extract the data necessary to create a [[ConversationControl]]

\* for a new [[Tweet]]. This is intended for use when creating

\* Tweets. It must be called after the Tweet has had its entities

\* extracted.

\*/

def fromTweet(

tweet: Tweet,

tweetCreateConversationControl: Option[TweetCreateConversationControl],

noteTweetMentionedUserIdsList: Option[Seq[Long]]

): Request = {

val cctl = tweetCreateConversationControl

new Request {

def tweetCreateConversationControl: Option[TweetCreateConversationControl] = cctl

def mentionedUserScreenNames: Set[ScreenName] =

tweet.mentions

// Enforce that the Tweet's mentions have already been

// extracted from the text. (Mentions will be None if they

// have not yet been extracted.)

.getOrElse(

throw new RuntimeException(

"Mentions must be extracted before applying ConversationControls"))

.map(\_.screenName)

.toSet

def tweetAuthorId: UserId = tweet.coreData.get.userId

def noteTweetMentionedUserIds: Option[Set[Long]] =

noteTweetMentionedUserIdsList.map(\_.toSet)

}

}

}

/\*\*

\* Create a ConversationControlBuilder that looks up user ids for

\* screen names using the specified UserIdentityRepository.

\*/

def fromUserIdentityRepo(

statsReceiver: StatsReceiver,

userIdentityRepo: UserIdentityRepository.Type

): Request => Stitch[Option[ConversationControl]] =

ConversationControlBuilder(

getUserId = screenName => userIdentityRepo(UserKey.byScreenName(screenName)).map(\_.id),

statsReceiver = statsReceiver

)

/\*\*

\* Extract the inviteViaMention value which does not exist on the TweetCreateConversationControl

\* itself but does exist on the structures it unions.

\*/

def inviteViaMention(tccc: TweetCreateConversationControl): Boolean =

tccc match {

case TweetCreateConversationControl.ByInvitation(c) => c.inviteViaMention.contains(true)

case TweetCreateConversationControl.Community(c) => c.inviteViaMention.contains(true)

case TweetCreateConversationControl.Followers(c) => c.inviteViaMention.contains(true)

case \_ => false

}

/\*\*

\* Translates the TweetCreateConversationControl into

\* ConversationControl using the context from the rest of the tweet

\* creation. For the most part, this is just a direct translation,

\* plus filling in the contextual user ids (mentioned users and tweet

\* author).

\*/

def apply(

statsReceiver: StatsReceiver,

getUserId: ScreenName => Stitch[UserId]

): Request => Stitch[Option[ConversationControl]] = {

val userIdLookupsCounter = statsReceiver.counter("user\_id\_lookups")

val conversationControlPresentCounter = statsReceiver.counter("conversation\_control\_present")

val conversationControlInviteViaMentionPresentCounter =

statsReceiver.counter("conversation\_control\_invite\_via\_mention\_present")

val failureCounter = statsReceiver.counter("failures")

// Get the user ids for these screen names. Any users who do not

// exist will be silently dropped.

def getExistingUserIds(

screenNames: Set[ScreenName],

mentionedUserIds: Option[Set[Long]]

): Stitch[Set[UserId]] = {

mentionedUserIds match {

case Some(userIds) => Stitch.value(userIds)

case \_ =>

Stitch

.traverse(screenNames.toSeq) { screenName =>

getUserId(screenName).liftNotFoundToOption

.ensure(userIdLookupsCounter.incr())

}

.map(userIdOptions => userIdOptions.flatten.toSet)

}

}

// This is broken out just to make it syntactically nicer to add

// the stats handling

def process(request: Request): Stitch[Option[ConversationControl]] =

request.tweetCreateConversationControl match {

case None => Stitch.None

case Some(cctl) =>

cctl match {

case TweetCreateConversationControl.ByInvitation(byInvitationControl) =>

for {

invitedUserIds <- getExistingUserIds(

request.mentionedUserScreenNames,

request.noteTweetMentionedUserIds)

} yield Some(

ConversationControls.byInvitation(

invitedUserIds = invitedUserIds.toSeq.filterNot(\_ == request.tweetAuthorId),

conversationTweetAuthorId = request.tweetAuthorId,

byInvitationControl.inviteViaMention

)

)

case TweetCreateConversationControl.Community(communityControl) =>

for {

invitedUserIds <- getExistingUserIds(

request.mentionedUserScreenNames,

request.noteTweetMentionedUserIds)

} yield Some(

ConversationControls.community(

invitedUserIds = invitedUserIds.toSeq.filterNot(\_ == request.tweetAuthorId),

conversationTweetAuthorId = request.tweetAuthorId,

communityControl.inviteViaMention

)

)

case TweetCreateConversationControl.Followers(followersControl) =>

for {

invitedUserIds <- getExistingUserIds(

request.mentionedUserScreenNames,

request.noteTweetMentionedUserIds)

} yield Some(

ConversationControls.followers(

invitedUserIds = invitedUserIds.toSeq.filterNot(\_ == request.tweetAuthorId),

conversationTweetAuthorId = request.tweetAuthorId,

followersControl.inviteViaMention

)

)

// This should only ever happen if a new value is added to the

// union and we don't update this code.

case TweetCreateConversationControl.UnknownUnionField(fld) =>

throw new RuntimeException(s"Unexpected TweetCreateConversationControl: $fld")

}

}

(request: Request) => {

// Wrap in Stitch to encapsulate any exceptions that happen

// before making a Stitch call inside of process.

Stitch(process(request)).flatten.respond { response =>

// If we count this before doing the work, and the stats are

// collected before the RPC completes, then any failures

// will get counted in a different minute than the request

// that caused it.

request.tweetCreateConversationControl.foreach { cc =>

conversationControlPresentCounter.incr()

if (inviteViaMention(cc)) conversationControlInviteViaMentionPresentCounter.incr()

}

response.onFailure { e =>

error(message = "Failed to create conversation control", cause = e)

// Don't bother counting individual exceptions, because

// the cost of keeping those stats is probably not worth

// the convenience of not having to look in the logs.

failureCounter.incr()

}

}

}

}

/\*\*

\* Validates if a conversation control request is allowed by feature switches

\* and is only requested on a root tweet.

\*/

object Validate {

case class Request(

matchedResults: Option[FeatureSwitchResults],

conversationControl: Option[TweetCreateConversationControl],

inReplyToTweetId: Option[TweetId])

type Type = FutureEffect[Request]

val ExInvalidConversationControl = TweetCreateFailure.State(InvalidConversationControl)

val ExConversationControlNotAllowed = TweetCreateFailure.State(ConversationControlNotAllowed)

val ConversationControlStatusUpdateEnabledKey = "conversation\_control\_status\_update\_enabled"

val ConversationControlFollowersEnabledKey = "conversation\_control\_my\_followers\_enabled"

def apply(

useFeatureSwitchResults: Gate[Unit],

statsReceiver: StatsReceiver

): Type = request => {

def fsDenied(fsKey: String): Boolean = {

val featureEnabledOpt: Option[Boolean] =

// Do not log impressions, which would interfere with shared client experiment data.

request.matchedResults.flatMap(\_.getBoolean(fsKey, shouldLogImpression = false))

val fsEnabled = featureEnabledOpt.contains(true)

if (!fsEnabled) {

statsReceiver.counter(s"check\_conversation\_control/unauthorized/fs/$fsKey").incr()

}

!fsEnabled

}

val isCcRequest: Boolean = request.conversationControl.isDefined

val isCcInvalidParams = isCcRequest && {

val isRootTweet = request.inReplyToTweetId.isEmpty

if (!isRootTweet) {

statsReceiver.counter("check\_conversation\_control/invalid").incr()

}

!isRootTweet

}

val isCcDeniedByFs = isCcRequest && {

val isFollower = request.conversationControl.exists {

case \_: TweetCreateConversationControl.Followers => true

case \_ => false

}

fsDenied(ConversationControlStatusUpdateEnabledKey) ||

(isFollower && fsDenied(ConversationControlFollowersEnabledKey))

}

if (isCcDeniedByFs && useFeatureSwitchResults()) {

Future.exception(ExConversationControlNotAllowed)

} else if (isCcInvalidParams) {

Future.exception(ExInvalidConversationControl)

} else {

Future.Unit

}

}

}

}