package com.twitter.tweetypie

package handler

import com.twitter.flockdb.client.\_

import com.twitter.snowflake.id.SnowflakeId

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.additionalfields.AdditionalFields.setAdditionalFields

import com.twitter.tweetypie.core.\_

import com.twitter.tweetypie.repository.\_

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.tweetypie.thriftscala.entities.EntityExtractor

import com.twitter.tweetypie.tweettext.Truncator

import com.twitter.tweetypie.util.CommunityUtil

import com.twitter.tweetypie.util.EditControlUtil

case class SourceTweetRequest(

tweetId: TweetId,

user: User,

hydrateOptions: WritePathHydrationOptions)

object RetweetBuilder {

import TweetBuilder.\_

import UpstreamFailure.\_

type Type = FutureArrow[RetweetRequest, TweetBuilderResult]

val SGSTestRole = "socialgraph"

val log: Logger = Logger(getClass)

/\*\*

\* Retweets text gets RT and username prepended

\*/

def composeRetweetText(text: String, sourceUser: User): String =

composeRetweetText(text, sourceUser.profile.get.screenName)

/\*\*

\* Retweets text gets RT and username prepended

\*/

def composeRetweetText(text: String, screenName: String): String =

Truncator.truncateForRetweet("RT @" + screenName + ": " + text)

// We do not want to allow community tweets to be retweeted.

def validateNotCommunityTweet(sourceTweet: Tweet): Future[Unit] =

if (CommunityUtil.hasCommunity(sourceTweet.communities)) {

Future.exception(TweetCreateFailure.State(TweetCreateState.CommunityRetweetNotAllowed))

} else {

Future.Unit

}

// We do not want to allow Trusted Friends tweets to be retweeted.

def validateNotTrustedFriendsTweet(sourceTweet: Tweet): Future[Unit] =

sourceTweet.trustedFriendsControl match {

case Some(trustedFriendsControl) =>

Future.exception(TweetCreateFailure.State(TweetCreateState.TrustedFriendsRetweetNotAllowed))

case None =>

Future.Unit

}

// We do not want to allow retweet of a stale version of a tweet in an edit chain.

def validateStaleTweet(sourceTweet: Tweet): Future[Unit] = {

if (!EditControlUtil.isLatestEdit(sourceTweet.editControl, sourceTweet.id).getOrElse(true)) {

Future.exception(TweetCreateFailure.State(TweetCreateState.StaleTweetRetweetNotAllowed))

} else {

// the source tweet does not have any edit control or the source tweet is the latest tweet

Future.Unit

}

}

/\*\*

\* Builds the RetweetBuilder

\*/

def apply(

validateRequest: RetweetRequest => Future[Unit],

tweetIdGenerator: TweetIdGenerator,

tweetRepo: TweetRepository.Type,

userRepo: UserRepository.Type,

tflock: TFlockClient,

deviceSourceRepo: DeviceSourceRepository.Type,

validateUpdateRateLimit: RateLimitChecker.Validate,

spamChecker: Spam.Checker[RetweetSpamRequest] = Spam.DoNotCheckSpam,

updateUserCounts: (User, Tweet) => Future[User],

superFollowRelationsRepo: StratoSuperFollowRelationsRepository.Type,

unretweetEdits: TweetDeletePathHandler.UnretweetEdits,

setEditWindowToSixtyMinutes: Gate[Unit]

): RetweetBuilder.Type = {

val entityExtactor = EntityExtractor.mutationAll.endo

val sourceTweetRepo: SourceTweetRequest => Stitch[Tweet] =

req => {

tweetRepo(

req.tweetId,

WritePathQueryOptions.retweetSourceTweet(req.user, req.hydrateOptions)

).rescue {

case \_: FilteredState => Stitch.NotFound

}

.rescue {

convertRepoExceptions(TweetCreateState.SourceTweetNotFound, TweetLookupFailure(\_))

}

}

val getUser = userLookup(userRepo)

val getSourceUser = sourceUserLookup(userRepo)

val getDeviceSource = deviceSourceLookup(deviceSourceRepo)

/\*\*

\* We exempt SGS test users from the check to get them through Block v2 testing.

\*/

def isSGSTestRole(user: User): Boolean =

user.roles.exists { roles => roles.roles.contains(SGSTestRole) }

def validateCanRetweet(

user: User,

sourceUser: User,

sourceTweet: Tweet,

request: RetweetRequest

): Future[Unit] =

Future

.join(

validateNotCommunityTweet(sourceTweet),

validateNotTrustedFriendsTweet(sourceTweet),

validateSourceUserRetweetable(user, sourceUser),

validateStaleTweet(sourceTweet),

Future.when(!request.dark) {

if (request.returnSuccessOnDuplicate)

failWithRetweetIdIfAlreadyRetweeted(user, sourceTweet)

else

validateNotAlreadyRetweeted(user, sourceTweet)

}

)

.unit

def validateSourceUserRetweetable(user: User, sourceUser: User): Future[Unit] =

if (sourceUser.profile.isEmpty)

Future.exception(UserProfileEmptyException)

else if (sourceUser.safety.isEmpty)

Future.exception(UserSafetyEmptyException)

else if (sourceUser.view.isEmpty)

Future.exception(UserViewEmptyException)

else if (user.id != sourceUser.id && sourceUser.safety.get.isProtected)

Future.exception(TweetCreateFailure.State(TweetCreateState.CannotRetweetProtectedTweet))

else if (sourceUser.safety.get.deactivated)

Future.exception(TweetCreateFailure.State(TweetCreateState.CannotRetweetDeactivatedUser))

else if (sourceUser.safety.get.suspended)

Future.exception(TweetCreateFailure.State(TweetCreateState.CannotRetweetSuspendedUser))

else if (sourceUser.view.get.blockedBy && !isSGSTestRole(user))

Future.exception(TweetCreateFailure.State(TweetCreateState.CannotRetweetBlockingUser))

else if (sourceUser.profile.get.screenName.isEmpty)

Future.exception(

TweetCreateFailure.State(TweetCreateState.CannotRetweetUserWithoutScreenName)

)

else

Future.Unit

def tflockGraphContains(

graph: StatusGraph,

fromId: Long,

toId: Long,

dir: Direction

): Future[Boolean] =

tflock.contains(graph, fromId, toId, dir).rescue {

case ex: OverCapacity => Future.exception(ex)

case ex => Future.exception(TFlockLookupFailure(ex))

}

def getRetweetIdFromTflock(sourceTweetId: TweetId, userId: UserId): Future[Option[Long]] =

tflock

.selectAll(

Select(

sourceId = sourceTweetId,

graph = RetweetsGraph,

direction = Forward

).intersect(

Select(

sourceId = userId,

graph = UserTimelineGraph,

direction = Forward

)

)

)

.map(\_.headOption)

def validateNotAlreadyRetweeted(user: User, sourceTweet: Tweet): Future[Unit] =

// use the perspective object from TLS if available, otherwise, check with tflock

(sourceTweet.perspective match {

case Some(perspective) =>

Future.value(perspective.retweeted)

case None =>

// we have to query the RetweetSourceGraph in the Reverse order because

// it is only defined in that direction, instead of bi-directionally

tflockGraphContains(RetweetSourceGraph, user.id, sourceTweet.id, Reverse)

}).flatMap {

case true =>

Future.exception(TweetCreateFailure.State(TweetCreateState.AlreadyRetweeted))

case false => Future.Unit

}

def failWithRetweetIdIfAlreadyRetweeted(user: User, sourceTweet: Tweet): Future[Unit] =

// use the perspective object from TLS if available, otherwise, check with tflock

(sourceTweet.perspective.flatMap(\_.retweetId) match {

case Some(tweetId) => Future.value(Some(tweetId))

case None =>

getRetweetIdFromTflock(sourceTweet.id, user.id)

}).flatMap {

case None => Future.Unit

case Some(tweetId) =>

Future.exception(TweetCreateFailure.AlreadyRetweeted(tweetId))

}

def validateContributor(contributorIdOpt: Option[UserId]): Future[Unit] =

if (contributorIdOpt.isDefined)

Future.exception(TweetCreateFailure.State(TweetCreateState.ContributorNotSupported))

else

Future.Unit

case class RetweetSource(sourceTweet: Tweet, parentUserId: UserId)

/\*\*

\* Recursively follows a retweet chain to the root source tweet. Also returns user id from the

\* first walked tweet as the 'parentUserId'.

\* In practice, the depth of the chain should never be greater than 2 because

\* share.sourceStatusId should always reference the root (unlike share.parentStatusId).

\*/

def findRetweetSource(

tweetId: TweetId,

forUser: User,

hydrateOptions: WritePathHydrationOptions

): Future[RetweetSource] =

Stitch

.run(sourceTweetRepo(SourceTweetRequest(tweetId, forUser, hydrateOptions)))

.flatMap { tweet =>

getShare(tweet) match {

case None => Future.value(RetweetSource(tweet, getUserId(tweet)))

case Some(share) =>

findRetweetSource(share.sourceStatusId, forUser, hydrateOptions)

.map(\_.copy(parentUserId = getUserId(tweet)))

}

}

FutureArrow { request =>

for {

() <- validateRequest(request)

userFuture = Stitch.run(getUser(request.userId))

tweetIdFuture = tweetIdGenerator()

devsrcFuture = Stitch.run(getDeviceSource(request.createdVia))

user <- userFuture

tweetId <- tweetIdFuture

devsrc <- devsrcFuture

rtSource <- findRetweetSource(

request.sourceStatusId,

user,

request.hydrationOptions.getOrElse(WritePathHydrationOptions(simpleQuotedTweet = true))

)

sourceTweet = rtSource.sourceTweet

sourceUser <- Stitch.run(getSourceUser(getUserId(sourceTweet), request.userId))

// We want to confirm that a user is actually allowed to

// retweet an Exclusive Tweet (only available to super followers)

() <- StratoSuperFollowRelationsRepository.Validate(

sourceTweet.exclusiveTweetControl,

user.id,

superFollowRelationsRepo)

() <- validateUser(user)

() <- validateUpdateRateLimit((user.id, request.dark))

() <- validateContributor(request.contributorUserId)

() <- validateCanRetweet(user, sourceUser, sourceTweet, request)

() <- unretweetEdits(sourceTweet.editControl, sourceTweet.id, user.id)

spamRequest = RetweetSpamRequest(

retweetId = tweetId,

sourceUserId = getUserId(sourceTweet),

sourceTweetId = sourceTweet.id,

sourceTweetText = getText(sourceTweet),

sourceUserName = sourceUser.profile.map(\_.screenName),

safetyMetaData = request.safetyMetaData

)

spamResult <- spamChecker(spamRequest)

safety = user.safety.get

share = Share(

sourceStatusId = sourceTweet.id,

sourceUserId = sourceUser.id,

parentStatusId = request.sourceStatusId

)

retweetText = composeRetweetText(getText(sourceTweet), sourceUser)

createdAt = SnowflakeId(tweetId).time

coreData = TweetCoreData(

userId = request.userId,

text = retweetText,

createdAtSecs = createdAt.inSeconds,

createdVia = devsrc.internalName,

share = Some(share),

hasTakedown = safety.hasTakedown,

trackingId = request.trackingId,

nsfwUser = safety.nsfwUser,

nsfwAdmin = safety.nsfwAdmin,

narrowcast = request.narrowcast,

nullcast = request.nullcast

)

retweet = Tweet(

id = tweetId,

coreData = Some(coreData),

contributor = getContributor(request.userId),

editControl = Some(

EditControl.Initial(

EditControlUtil

.makeEditControlInitial(

tweetId = tweetId,

createdAt = createdAt,

setEditWindowToSixtyMinutes = setEditWindowToSixtyMinutes

)

.initial

.copy(isEditEligible = Some(false))

)

),

)

retweetWithEntities = entityExtactor(retweet)

retweetWithAdditionalFields = setAdditionalFields(

retweetWithEntities,

request.additionalFields

)

// update the perspective and counts fields of the source tweet to reflect the effects

// of the user performing a retweet, even though those effects haven't happened yet.

updatedSourceTweet = sourceTweet.copy(

perspective = sourceTweet.perspective.map {

\_.copy(retweeted = true, retweetId = Some(retweet.id))

},

counts = sourceTweet.counts.map { c => c.copy(retweetCount = c.retweetCount.map(\_ + 1)) }

)

user <- updateUserCounts(user, retweetWithAdditionalFields)

} yield {

TweetBuilderResult(

tweet = retweetWithAdditionalFields,

user = user,

createdAt = createdAt,

sourceTweet = Some(updatedSourceTweet),

sourceUser = Some(sourceUser),

parentUserId = Some(rtSource.parentUserId),

isSilentFail = spamResult == Spam.SilentFail

)

}

}

}

}