package com.twitter.tweetypie

package handler

import com.twitter.conversions.DurationOps.RichDuration

import com.twitter.servo.exception.thriftscala.ClientError

import com.twitter.servo.exception.thriftscala.ClientErrorCause

import com.twitter.servo.util.FutureArrow

import com.twitter.snowflake.id.SnowflakeId

import com.twitter.stitch.Stitch

import com.twitter.stitch.NotFound

import com.twitter.timelineservice.thriftscala.PerspectiveResult

import com.twitter.timelineservice.{thriftscala => tls}

import com.twitter.tweetypie.core.\_

import com.twitter.tweetypie.repository.\_

import com.twitter.tweetypie.store.\_

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.util.Time

import com.twitter.util.Try

import Try.\_

import com.twitter.spam.rtf.thriftscala.SafetyLabelType

import com.twitter.tweetypie.backends.TimelineService.GetPerspectives

import com.twitter.tweetypie.util.EditControlUtil

import scala.util.control.NoStackTrace

case class CascadedDeleteNotAvailable(retweetId: TweetId) extends Exception with NoStackTrace {

override def getMessage: String =

s"""|Cascaded delete tweet failed because tweet $retweetId

|is not present in cache or manhattan.""".stripMargin

}

object TweetDeletePathHandler {

type DeleteTweets =

(DeleteTweetsRequest, Boolean) => Future[Seq[DeleteTweetResult]]

type UnretweetEdits = (Option[EditControl], TweetId, UserId) => Future[Unit]

/\*\* The information from a deleteTweet request that can be inspected by a deleteTweets validator \*/

case class DeleteTweetsContext(

byUserId: Option[UserId],

authenticatedUserId: Option[UserId],

tweetAuthorId: UserId,

users: Map[UserId, User],

isUserErasure: Boolean,

expectedErasureUserId: Option[UserId],

tweetIsBounced: Boolean,

isBounceDelete: Boolean)

/\*\* Provides reason a tweet deletion was allowed \*/

sealed trait DeleteAuthorization { def byUserId: Option[UserId] }

case class AuthorizedByTweetOwner(userId: UserId) extends DeleteAuthorization {

def byUserId: Option[UserId] = Some(userId)

}

case class AuthorizedByTweetContributor(contributorUserId: UserId) extends DeleteAuthorization {

def byUserId: Option[UserId] = Some(contributorUserId)

}

case class AuthorizedByAdmin(adminUserId: UserId) extends DeleteAuthorization {

def byUserId: Option[UserId] = Some(adminUserId)

}

case object AuthorizedByErasure extends DeleteAuthorization {

def byUserId: None.type = None

}

// Type for a method that receives all the relevant information about a proposed internal tweet

// deletion and can return Future.exception to cancel the delete due to a validation error or

// return a [[DeleteAuthorization]] specifying the reason the deletion is allowed.

type ValidateDeleteTweets = FutureArrow[DeleteTweetsContext, DeleteAuthorization]

val userFieldsForDelete: Set[UserField] =

Set(UserField.Account, UserField.Profile, UserField.Roles, UserField.Safety)

val userQueryOptions: UserQueryOptions =

UserQueryOptions(

userFieldsForDelete,

UserVisibility.All

)

// user\_agent property originates from the client so truncate to a reasonable length

val MaxUserAgentLength = 1000

// Age under which we treat not found tweets in

// cascaded\_delete\_tweet as a temporary condition (the most likely

// explanation being that the tweet has not yet been

// replicated). Tweets older than this we assume are due to

// \*permanently\* inconsistent data, either spurious edges in tflock or

// tweets that are not loadable from Manhattan.

val MaxCascadedDeleteTweetTemporaryInconsistencyAge: Duration =

10.minutes

}

trait TweetDeletePathHandler {

import TweetDeletePathHandler.ValidateDeleteTweets

def cascadedDeleteTweet(request: CascadedDeleteTweetRequest): Future[Unit]

def deleteTweets(

request: DeleteTweetsRequest,

isUnretweetEdits: Boolean = false,

): Future[Seq[DeleteTweetResult]]

def internalDeleteTweets(

request: DeleteTweetsRequest,

byUserId: Option[UserId],

authenticatedUserId: Option[UserId],

validate: ValidateDeleteTweets,

isUnretweetEdits: Boolean = false

): Future[Seq[DeleteTweetResult]]

def unretweetEdits(

optEditControl: Option[EditControl],

excludedTweetId: TweetId,

byUserId: UserId

): Future[Unit]

}

/\*\*

\* Implementation of TweetDeletePathHandler

\*/

class DefaultTweetDeletePathHandler(

stats: StatsReceiver,

tweetResultRepo: TweetResultRepository.Type,

userRepo: UserRepository.Optional,

stratoSafetyLabelsRepo: StratoSafetyLabelsRepository.Type,

lastQuoteOfQuoterRepo: LastQuoteOfQuoterRepository.Type,

tweetStore: TotalTweetStore,

getPerspectives: GetPerspectives)

extends TweetDeletePathHandler {

import TweetDeletePathHandler.\_

val tweetRepo: TweetRepository.Type = TweetRepository.fromTweetResult(tweetResultRepo)

// attempt to delete tweets was made by someone other than the tweet owner or an admin user

object DeleteTweetsPermissionException extends Exception with NoStackTrace

object ExpectedUserIdMismatchException extends Exception with NoStackTrace

private[this] val log = Logger("com.twitter.tweetypie.store.TweetDeletions")

private[this] val cascadeEditDelete = stats.scope("cascade\_edit\_delete")

private[this] val cascadeEditDeletesEnqueued = cascadeEditDelete.counter("enqueued")

private[this] val cascadeEditDeleteTweets = cascadeEditDelete.counter("tweets")

private[this] val cascadeEditDeleteFailures = cascadeEditDelete.counter("failures")

private[this] val cascadedDeleteTweet = stats.scope("cascaded\_delete\_tweet")

private[this] val cascadedDeleteTweetFailures = cascadedDeleteTweet.counter("failures")

private[this] val cascadedDeleteTweetSourceMatch = cascadedDeleteTweet.counter("source\_match")

private[this] val cascadedDeleteTweetSourceMismatch =

cascadedDeleteTweet.counter("source\_mismatch")

private[this] val cascadedDeleteTweetTweetNotFound =

cascadedDeleteTweet.counter("tweet\_not\_found")

private[this] val cascadedDeleteTweetTweetNotFoundAge =

cascadedDeleteTweet.stat("tweet\_not\_found\_age")

private[this] val cascadedDeleteTweetUserNotFound = cascadedDeleteTweet.counter("user\_not\_found")

private[this] val deleteTweets = stats.scope("delete\_tweets")

private[this] val deleteTweetsAuth = deleteTweets.scope("per\_tweet\_auth")

private[this] val deleteTweetsAuthAttempts = deleteTweetsAuth.counter("attempts")

private[this] val deleteTweetsAuthFailures = deleteTweetsAuth.counter("failures")

private[this] val deleteTweetsAuthSuccessAdmin = deleteTweetsAuth.counter("success\_admin")

private[this] val deleteTweetsAuthSuccessByUser = deleteTweetsAuth.counter("success\_by\_user")

private[this] val deleteTweetsTweets = deleteTweets.counter("tweets")

private[this] val deleteTweetsFailures = deleteTweets.counter("failures")

private[this] val deleteTweetsTweetNotFound = deleteTweets.counter("tweet\_not\_found")

private[this] val deleteTweetsUserNotFound = deleteTweets.counter("user\_not\_found")

private[this] val userIdMismatchInTweetDelete =

deleteTweets.counter("expected\_actual\_user\_id\_mismatch")

private[this] val bounceDeleteFlagNotSet =

deleteTweets.counter("bounce\_delete\_flag\_not\_set")

private[this] def getUser(userId: UserId): Future[Option[User]] =

Stitch.run(userRepo(UserKey(userId), userQueryOptions))

private[this] def getUsersForDeleteTweets(userIds: Seq[UserId]): Future[Map[UserId, User]] =

Stitch.run(

Stitch

.traverse(userIds) { userId =>

userRepo(UserKey(userId), userQueryOptions).map {

case Some(u) => Some(userId -> u)

case None => deleteTweetsUserNotFound.incr(); None

}

}

.map(\_.flatten.toMap)

)

private[this] def getTweet(tweetId: TweetId): Future[Tweet] =

Stitch.run(tweetRepo(tweetId, WritePathQueryOptions.deleteTweetsWithoutEditControl))

private[this] def getSingleDeletedTweet(

id: TweetId,

isCascadedEditTweetDeletion: Boolean = false

): Stitch[Option[TweetData]] = {

val opts = if (isCascadedEditTweetDeletion) {

// Disable edit control hydration if this is cascade delete of edits.

// When edit control is hydrated, the tweet will actually be considered already deleted.

WritePathQueryOptions.deleteTweetsWithoutEditControl

} else {

WritePathQueryOptions.deleteTweets

}

tweetResultRepo(id, opts)

.map(\_.value)

.liftToOption {

// We treat the request the same whether the tweet never

// existed or is in one of the already-deleted states by

// just filtering out those tweets. Any tweets that we

// return should be deleted. If the tweet has been

// bounce-deleted, we never want to soft-delete it, and

// vice versa.

case NotFound | FilteredState.Unavailable.TweetDeleted |

FilteredState.Unavailable.BounceDeleted =>

true

}

}

private[this] def getTweetsForDeleteTweets(

ids: Seq[TweetId],

isCascadedEditTweetDeletion: Boolean

): Future[Map[TweetId, TweetData]] =

Stitch

.run {

Stitch.traverse(ids) { id =>

getSingleDeletedTweet(id, isCascadedEditTweetDeletion)

.map {

// When deleting a tweet that has been edited, we want to instead delete the initial version.

// Because the initial tweet will be hydrated in every request, if it is deleted, later

// revisions will be hidden, and cleaned up asynchronously by TP Daemons

// However, we don't need to do a second lookup if it's already the original tweet

// or if we're doing a cascading edit tweet delete (deleting the entire tweet history)

case Some(tweetData)

if EditControlUtil.isInitialTweet(tweetData.tweet) ||

isCascadedEditTweetDeletion =>

Stitch.value(Some(tweetData))

case Some(tweetData) =>

getSingleDeletedTweet(EditControlUtil.getInitialTweetId(tweetData.tweet))

case None =>

Stitch.value(None)

// We need to preserve the input tweetId, and the initial TweetData

}.flatten.map(tweetData => (id, tweetData))

}

}

.map(\_.collect { case (tweetId, Some(tweetData)) => (tweetId, tweetData) }.toMap)

private[this] def getStratoBounceStatuses(

ids: Seq[Long],

isUserErasure: Boolean,

isCascadedEditedTweetDeletion: Boolean

): Future[Map[TweetId, Boolean]] = {

// Don't load bounce label for user erasure tweet deletion.

// User Erasure deletions cause unnecessary spikes of traffic

// to Strato when we read the bounce label that we don't use.

// We also want to always delete a bounced tweet if the rest of the

// edit chain is being deleted in a cascaded edit tweet delete

if (isUserErasure || isCascadedEditedTweetDeletion) {

Future.value(ids.map(id => id -> false).toMap)

} else {

Stitch.run(

Stitch

.traverse(ids) { id =>

stratoSafetyLabelsRepo(id, SafetyLabelType.Bounce).map { label =>

id -> label.isDefined

}

}

.map(\_.toMap)

)

}

}

/\*\* A suspended/deactivated user can't delete tweets \*/

private[this] def userNotSuspendedOrDeactivated(user: User): Try[User] =

user.safety match {

case None => Throw(UpstreamFailure.UserSafetyEmptyException)

case Some(safety) if safety.deactivated =>

Throw(

AccessDenied(

s"User deactivated userId: ${user.id}",

errorCause = Some(AccessDeniedCause.UserDeactivated)

)

)

case Some(safety) if safety.suspended =>

Throw(

AccessDenied(

s"User suspended userId: ${user.id}",

errorCause = Some(AccessDeniedCause.UserSuspended)

)

)

case \_ => Return(user)

}

/\*\*

\* Ensure that byUser has permission to delete tweet either by virtue of owning the tweet or being

\* an admin user. Returns the reason as a DeleteAuthorization or else throws an Exception if not

\* authorized.

\*/

private[this] def userAuthorizedToDeleteTweet(

byUser: User,

optAuthenticatedUserId: Option[UserId],

tweetAuthorId: UserId

): Try[DeleteAuthorization] = {

def hasAdminPrivilege =

byUser.roles.exists(\_.rights.contains("delete\_user\_tweets"))

deleteTweetsAuthAttempts.incr()

if (byUser.id == tweetAuthorId) {

deleteTweetsAuthSuccessByUser.incr()

optAuthenticatedUserId match {

case Some(uid) =>

Return(AuthorizedByTweetContributor(uid))

case None =>

Return(AuthorizedByTweetOwner(byUser.id))

}

} else if (optAuthenticatedUserId.isEmpty && hasAdminPrivilege) { // contributor may not assume admin role

deleteTweetsAuthSuccessAdmin.incr()

Return(AuthorizedByAdmin(byUser.id))

} else {

deleteTweetsAuthFailures.incr()

Throw(DeleteTweetsPermissionException)

}

}

/\*\*

\* expected user id is the id provided on the DeleteTweetsRequest that the indicates which user

\* owns the tweets they want to delete. The actualUserId is the actual userId on the tweet we are about to delete.

\* we check to ensure they are the same as a safety check against accidental deletion of tweets either from user mistakes

\* or from corrupted data (e.g bad tflock edges)

\*/

private[this] def expectedUserIdMatchesActualUserId(

expectedUserId: UserId,

actualUserId: UserId

): Try[Unit] =

if (expectedUserId == actualUserId) {

Return.Unit

} else {

userIdMismatchInTweetDelete.incr()

Throw(ExpectedUserIdMismatchException)

}

/\*\*

\* Validation for the normal public tweet delete case, the user must be found and must

\* not be suspended or deactivated.

\*/

val validateTweetsForPublicDelete: ValidateDeleteTweets = FutureArrow {

ctx: DeleteTweetsContext =>

Future.const(

for {

// byUserId must be present

byUserId <- ctx.byUserId.orThrow(

ClientError(ClientErrorCause.BadRequest, "Missing byUserId")

)

// the byUser must be found

byUserOpt = ctx.users.get(byUserId)

byUser <- byUserOpt.orThrow(

ClientError(ClientErrorCause.BadRequest, s"User $byUserId not found")

)

\_ <- userNotSuspendedOrDeactivated(byUser)

\_ <- validateBounceConditions(

ctx.tweetIsBounced,

ctx.isBounceDelete

)

// if there's a contributor, make sure the user is found and not suspended or deactivated

\_ <-

ctx.authenticatedUserId

.map { uid =>

ctx.users.get(uid) match {

case None =>

Throw(ClientError(ClientErrorCause.BadRequest, s"Contributor $uid not found"))

case Some(authUser) =>

userNotSuspendedOrDeactivated(authUser)

}

}

.getOrElse(Return.Unit)

// if the expected user id is present, make sure it matches the user id on the tweet

\_ <-

ctx.expectedErasureUserId

.map { expectedUserId =>

expectedUserIdMatchesActualUserId(expectedUserId, ctx.tweetAuthorId)

}

.getOrElse(Return.Unit)

// User must own the tweet or be an admin

deleteAuth <- userAuthorizedToDeleteTweet(

byUser,

ctx.authenticatedUserId,

ctx.tweetAuthorId

)

} yield deleteAuth

)

}

private def validateBounceConditions(

tweetIsBounced: Boolean,

isBounceDelete: Boolean

): Try[Unit] = {

if (tweetIsBounced && !isBounceDelete) {

bounceDeleteFlagNotSet.incr()

Throw(ClientError(ClientErrorCause.BadRequest, "Cannot normal delete a Bounced Tweet"))

} else {

Return.Unit

}

}

/\*\*

\* Validation for the user erasure case. User may be missing.

\*/

val validateTweetsForUserErasureDaemon: ValidateDeleteTweets = FutureArrow {

ctx: DeleteTweetsContext =>

Future

.const(

for {

expectedUserId <- ctx.expectedErasureUserId.orThrow(

ClientError(

ClientErrorCause.BadRequest,

"expectedUserId is required for DeleteTweetRequests"

)

)

// It's critical to always check that the userId on the tweet we want to delete matches the

// userId on the erasure request. This prevents us from accidentally deleting tweets not owned by the

// erased user, even if tflock serves us bad data.

validationResult <- expectedUserIdMatchesActualUserId(expectedUserId, ctx.tweetAuthorId)

} yield validationResult

)

.map(\_ => AuthorizedByErasure)

}

/\*\*

\* Fill in missing values of AuditDeleteTweet with values from TwitterContext.

\*/

def enrichMissingFromTwitterContext(orig: AuditDeleteTweet): AuditDeleteTweet = {

val viewer = TwitterContext()

orig.copy(

host = orig.host.orElse(viewer.flatMap(\_.auditIp)),

clientApplicationId = orig.clientApplicationId.orElse(viewer.flatMap(\_.clientApplicationId)),

userAgent = orig.userAgent.orElse(viewer.flatMap(\_.userAgent)).map(\_.take(MaxUserAgentLength))

)

}

/\*\*

\* core delete tweets implementation.

\*

\* The [[deleteTweets]] method wraps this method and provides validation required

\* for a public endpoint.

\*/

override def internalDeleteTweets(

request: DeleteTweetsRequest,

byUserId: Option[UserId],

authenticatedUserId: Option[UserId],

validate: ValidateDeleteTweets,

isUnretweetEdits: Boolean = false

): Future[Seq[DeleteTweetResult]] = {

val auditDeleteTweet =

enrichMissingFromTwitterContext(request.auditPassthrough.getOrElse(AuditDeleteTweet()))

deleteTweetsTweets.incr(request.tweetIds.size)

for {

tweetDataMap <- getTweetsForDeleteTweets(

request.tweetIds,

request.cascadedEditedTweetDeletion.getOrElse(false)

)

userIds: Seq[UserId] = (tweetDataMap.values.map { td =>

getUserId(td.tweet)

} ++ byUserId ++ authenticatedUserId).toSeq.distinct

users <- getUsersForDeleteTweets(userIds)

stratoBounceStatuses <- getStratoBounceStatuses(

tweetDataMap.keys.toSeq,

request.isUserErasure,

request.cascadedEditedTweetDeletion.getOrElse(false))

results <- Future.collect {

request.tweetIds.map { tweetId =>

tweetDataMap.get(tweetId) match {

// already deleted, so nothing to do

case None =>

deleteTweetsTweetNotFound.incr()

Future.value(DeleteTweetResult(tweetId, TweetDeleteState.Ok))

case Some(tweetData) =>

val tweet: Tweet = tweetData.tweet

val tweetIsBounced = stratoBounceStatuses(tweetId)

val optSourceTweet: Option[Tweet] = tweetData.sourceTweetResult.map(\_.value.tweet)

val validation: Future[(Boolean, DeleteAuthorization)] = for {

isLastQuoteOfQuoter <- isFinalQuoteOfQuoter(tweet)

deleteAuth <- validate(

DeleteTweetsContext(

byUserId = byUserId,

authenticatedUserId = authenticatedUserId,

tweetAuthorId = getUserId(tweet),

users = users,

isUserErasure = request.isUserErasure,

expectedErasureUserId = request.expectedUserId,

tweetIsBounced = tweetIsBounced,

isBounceDelete = request.isBounceDelete

)

)

\_ <- optSourceTweet match {

case Some(sourceTweet) if !isUnretweetEdits =>

// If this is a retweet and this deletion was not triggered by

// unretweetEdits, unretweet edits of the source Tweet

// before deleting the retweet.

//

// deleteAuth will always contain a byUserId except for erasure deletion,

// in which case the retweets will be deleted individually.

deleteAuth.byUserId match {

case Some(userId) =>

unretweetEdits(sourceTweet.editControl, sourceTweet.id, userId)

case None => Future.Unit

}

case \_ => Future.Unit

}

} yield {

(isLastQuoteOfQuoter, deleteAuth)

}

validation

.flatMap {

case (isLastQuoteOfQuoter: Boolean, deleteAuth: DeleteAuthorization) =>

val isAdminDelete = deleteAuth match {

case AuthorizedByAdmin(\_) => true

case \_ => false

}

val event =

DeleteTweet.Event(

tweet = tweet,

timestamp = Time.now,

user = users.get(getUserId(tweet)),

byUserId = deleteAuth.byUserId,

auditPassthrough = Some(auditDeleteTweet),

isUserErasure = request.isUserErasure,

isBounceDelete = request.isBounceDelete && tweetIsBounced,

isLastQuoteOfQuoter = isLastQuoteOfQuoter,

isAdminDelete = isAdminDelete

)

val numberOfEdits: Int = tweet.editControl

.collect {

case EditControl.Initial(initial) =>

initial.editTweetIds.count(\_ != tweet.id)

}

.getOrElse(0)

cascadeEditDeletesEnqueued.incr(numberOfEdits)

tweetStore

.deleteTweet(event)

.map(\_ => DeleteTweetResult(tweetId, TweetDeleteState.Ok))

}

.onFailure { \_ =>

deleteTweetsFailures.incr()

}

.handle {

case ExpectedUserIdMismatchException =>

DeleteTweetResult(tweetId, TweetDeleteState.ExpectedUserIdMismatch)

case DeleteTweetsPermissionException =>

DeleteTweetResult(tweetId, TweetDeleteState.PermissionError)

}

}

}

}

} yield results

}

private def isFinalQuoteOfQuoter(tweet: Tweet): Future[Boolean] = {

tweet.quotedTweet match {

case Some(qt) =>

Stitch.run {

lastQuoteOfQuoterRepo

.apply(qt.tweetId, getUserId(tweet))

.liftToTry

.map(\_.getOrElse(false))

}

case None => Future(false)

}

}

/\*\*

\* Validations for the public deleteTweets endpoint.

\* - ensures that the byUserId user can be found and is in the correct user state

\* - ensures that the tweet is being deleted by the tweet's owner, or by an admin

\* If there is a validation error, a future.exception is returned

\*

\* If the delete request is part of a user erasure, validations are relaxed (the User is allowed to be missing).

\*/

val deleteTweetsValidator: ValidateDeleteTweets =

FutureArrow { context =>

if (context.isUserErasure) {

validateTweetsForUserErasureDaemon(context)

} else {

validateTweetsForPublicDelete(context)

}

}

override def deleteTweets(

request: DeleteTweetsRequest,

isUnretweetEdits: Boolean = false,

): Future[Seq[DeleteTweetResult]] = {

// For comparison testing we only want to compare the DeleteTweetsRequests that are generated

// in DeleteTweets path and not the call that comes from the Unretweet path

val context = TwitterContext()

internalDeleteTweets(

request,

byUserId = request.byUserId.orElse(context.flatMap(\_.userId)),

context.flatMap(\_.authenticatedUserId),

deleteTweetsValidator,

isUnretweetEdits

)

}

// Cascade delete tweet is the logic for removing tweets that are detached

// from their dependency which has been deleted. They are already filtered

// out from serving, so this operation reconciles storage with the view

// presented by Tweetypie.

// This RPC call is delegated from daemons or batch jobs. Currently there

// are two use-cases when this call is issued:

// \* Deleting detached retweets after the source tweet was deleted.

// This is done through RetweetsDeletion daemon and the

// CleanupDetachedRetweets job.

// \* Deleting edits of an initial tweet that has been deleted.

// This is done by CascadedEditedTweetDelete daemon.

// Note that, when serving the original delete request for an edit,

// the initial tweet is only deleted, which makes all edits hidden.

override def cascadedDeleteTweet(request: CascadedDeleteTweetRequest): Future[Unit] = {

val contextViewer = TwitterContext()

getTweet(request.tweetId)

.transform {

case Throw(

FilteredState.Unavailable.TweetDeleted | FilteredState.Unavailable.BounceDeleted) =>

// The retweet or edit was already deleted via some other mechanism

Future.Unit

case Throw(NotFound) =>

cascadedDeleteTweetTweetNotFound.incr()

val recentlyCreated =

if (SnowflakeId.isSnowflakeId(request.tweetId)) {

val age = Time.now - SnowflakeId(request.tweetId).time

cascadedDeleteTweetTweetNotFoundAge.add(age.inMilliseconds)

age < MaxCascadedDeleteTweetTemporaryInconsistencyAge

} else {

false

}

if (recentlyCreated) {

// Treat the NotFound as a temporary condition, most

// likely due to replication lag.

Future.exception(CascadedDeleteNotAvailable(request.tweetId))

} else {

// Treat the NotFound as a permanent inconsistenty, either

// spurious edges in tflock or invalid data in Manhattan. This

// was happening a few times an hour during the time that we

// were not treating it specially. For now, we will just log that

// it happened, but in the longer term, it would be good

// to collect this data and repair the corruption.

log.warn(

Seq(

"cascaded\_delete\_tweet\_old\_not\_found",

request.tweetId,

request.cascadedFromTweetId

).mkString("\t")

)

Future.Done

}

// Any other FilteredStates should not be thrown because of

// the options that we used to load the tweet, so we will just

// let them bubble up as an internal server error

case Throw(other) =>

Future.exception(other)

case Return(tweet) =>

Future

.join(

isFinalQuoteOfQuoter(tweet),

getUser(getUserId(tweet))

)

.flatMap {

case (isLastQuoteOfQuoter, user) =>

if (user.isEmpty) {

cascadedDeleteTweetUserNotFound.incr()

}

val tweetSourceId = getShare(tweet).map(\_.sourceStatusId)

val initialEditId = tweet.editControl.collect {

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

}

if (initialEditId.contains(request.cascadedFromTweetId)) {

cascadeEditDeleteTweets.incr()

}

if (tweetSourceId.contains(request.cascadedFromTweetId)

|| initialEditId.contains(request.cascadedFromTweetId)) {

cascadedDeleteTweetSourceMatch.incr()

val deleteEvent =

DeleteTweet.Event(

tweet = tweet,

timestamp = Time.now,

user = user,

byUserId = contextViewer.flatMap(\_.userId),

cascadedFromTweetId = Some(request.cascadedFromTweetId),

auditPassthrough = request.auditPassthrough,

isUserErasure = false,

// cascaded deletes of retweets or edits have not been through a bouncer flow,

// so are not considered to be "bounce deleted".

isBounceDelete = false,

isLastQuoteOfQuoter = isLastQuoteOfQuoter,

isAdminDelete = false

)

tweetStore

.deleteTweet(deleteEvent)

.onFailure { \_ =>

if (initialEditId.contains(request.cascadedFromTweetId)) {

cascadeEditDeleteFailures.incr()

}

}

} else {

cascadedDeleteTweetSourceMismatch.incr()

log.warn(

Seq(

"cascaded\_from\_tweet\_id\_source\_mismatch",

request.tweetId,

request.cascadedFromTweetId,

tweetSourceId.orElse(initialEditId).getOrElse("-")

).mkString("\t")

)

Future.Done

}

}

}

.onFailure(\_ => cascadedDeleteTweetFailures.incr())

}

// Given a list of edit Tweet ids and a user id, find the retweet ids of those edit ids from the given user

private def editTweetIdRetweetsFromUser(

editTweetIds: Seq[TweetId],

byUserId: UserId

): Future[Seq[TweetId]] = {

if (editTweetIds.isEmpty) {

Future.value(Seq())

} else {

getPerspectives(

Seq(tls.PerspectiveQuery(byUserId, editTweetIds))

).map { res: Seq[PerspectiveResult] =>

res.headOption.toSeq

.flatMap(\_.perspectives.flatMap(\_.retweetId))

}

}

}

/\* This function is called from three places -

\* 1. When Tweetypie gets a request to retweet the latest version of an edit chain, all the

\* previous revisons should be unretweeted.

\* i.e. On Retweet of the latest tweet - unretweets all the previous revisions for this user.

\* - create A

\* - retweet A'(retweet of A)

\* - create edit B(edit of A)

\* - retweet B' => Deletes A'

\*

\* 2. When Tweetypie gets an unretweet request for a source tweet that is an edit tweet, all

\* the versions of the edit chain is retweeted.

\* i.e. On unretweet of any version in the edit chain - unretweets all the revisions for this user

\* - create A

\* - retweet A'

\* - create B

\* - unretweet B => Deletes A' (& also any B' if it existed)

\*

\* 3. When Tweetypie gets a delete request for a retweet, say A1. & if A happens to the source

\* tweet for A1 & if A is an edit tweet, then the entire edit chain should be unretweeted & not

\* A. i.e. On delete of a retweet - unretweet all the revisions for this user.

\* - create A

\* - retweet A'

\* - create B

\* - delete A' => Deletes A' (& also any B' if it existed)

\*

\* The following function has two failure scenarios -

\* i. when it fails to get perspectives of any of the edit tweets.

\* ii. the deletion of any of the retweets of these edits fail.

\*

\* In either of this scenario, we fail the entire request & the error bubbles up to the top.

\* Note: The above unretweet of edits only happens for the current user.

\* In normal circumstances, a maximum of one Tweet in the edit chain will have been retweeted,

\* but we don't know which one it was. Additionally, there may be circumstances where

\* unretweet failed, and we end up with multiple versions retweeted. For these reasons,

\* we always unretweet all the revisions (except for `excludedTweetId`).

\* This is a no-op if none of these versions have been retweeted.

\* \*/

override def unretweetEdits(

optEditControl: Option[EditControl],

excludedTweetId: TweetId,

byUserId: UserId

): Future[Unit] = {

val editTweetIds: Seq[TweetId] =

EditControlUtil.getEditTweetIds(optEditControl).get().filter(\_ != excludedTweetId)

(editTweetIdRetweetsFromUser(editTweetIds, byUserId).flatMap { tweetIds =>

if (tweetIds.nonEmpty) {

deleteTweets(

DeleteTweetsRequest(tweetIds = tweetIds, byUserId = Some(byUserId)),

isUnretweetEdits = true

)

} else {

Future.Nil

}

}).unit

}

}