package com.twitter.tweetypie

package handler

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.core.TweetCreateFailure

import com.twitter.tweetypie.repository.\_

import com.twitter.tweetypie.serverutil.ExceptionCounter

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.tweetypie.tweettext.Offset

import com.twitter.twittertext.Extractor

import scala.annotation.tailrec

import scala.collection.JavaConverters.\_

import scala.collection.mutable

import scala.util.control.NoStackTrace

object ReplyBuilder {

private val extractor = new Extractor

private val InReplyToTweetNotFound =

TweetCreateFailure.State(TweetCreateState.InReplyToTweetNotFound)

case class Request(

authorId: UserId,

authorScreenName: String,

inReplyToTweetId: Option[TweetId],

tweetText: String,

prependImplicitMentions: Boolean,

enableTweetToNarrowcasting: Boolean,

excludeUserIds: Seq[UserId],

spamResult: Spam.Result,

batchMode: Option[BatchComposeMode])

/\*\*

\* This case class contains the fields that are shared between legacy and simplified replies.

\*/

case class BaseResult(

reply: Reply,

conversationId: Option[ConversationId],

selfThreadMetadata: Option[SelfThreadMetadata],

community: Option[Communities] = None,

exclusiveTweetControl: Option[ExclusiveTweetControl] = None,

trustedFriendsControl: Option[TrustedFriendsControl] = None,

editControl: Option[EditControl] = None) {

// Creates a Result by providing the fields that differ between legacy and simplified replies.

def toResult(

tweetText: String,

directedAtMetadata: DirectedAtUserMetadata,

visibleStart: Offset.CodePoint = Offset.CodePoint(0),

): Result =

Result(

reply,

tweetText,

directedAtMetadata,

conversationId,

selfThreadMetadata,

visibleStart,

community,

exclusiveTweetControl,

trustedFriendsControl,

editControl

)

}

/\*\*

\* @param reply the Reply object to include in the tweet.

\* @param tweetText updated tweet text which may include prepended at-mentions, trimmed

\* @param directedAtMetadata see DirectedAtHydrator for usage.

\* @param conversationId conversation id to assign to the tweet.

\* @param selfThreadMetadata returns the result of `SelfThreadBuilder`

\* @param visibleStart offset into `tweetText` separating hideable at-mentions from the

\* visible text.

\*/

case class Result(

reply: Reply,

tweetText: String,

directedAtMetadata: DirectedAtUserMetadata,

conversationId: Option[ConversationId] = None,

selfThreadMetadata: Option[SelfThreadMetadata] = None,

visibleStart: Offset.CodePoint = Offset.CodePoint(0),

community: Option[Communities] = None,

exclusiveTweetControl: Option[ExclusiveTweetControl] = None,

trustedFriendsControl: Option[TrustedFriendsControl] = None,

editControl: Option[EditControl] = None) {

/\*\*

\* @param finalText final tweet text after any server-side additions.

\* @return true iff the final tweet text consists exclusively of a hidden reply mention prefix.

\* When this happens there's no content to the reply and thus the tweet creation should

\* fail.

\*/

def replyTextIsEmpty(finalText: String): Boolean = {

// Length of the tweet text originally output via ReplyBuilder.Result before server-side

// additions (e.g. media, quoted-tweet URLs)

val origTextLength = Offset.CodePoint.length(tweetText)

// Length of the tweet text after server-side additions.

val finalTextLength = Offset.CodePoint.length(finalText)

val prefixWasEntireText = origTextLength == visibleStart

val textLenUnchanged = origTextLength == finalTextLength

prefixWasEntireText && textLenUnchanged

}

}

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

private object InvalidUserException extends NoStackTrace

/\*\*

\* A user ID and screen name used for building replies.

\*/

private case class User(id: UserId, screenName: String)

/\*\*

\* Captures the in-reply-to tweet, its author, and if the user is attempting to reply to a

\* retweet, then that retweet and its author.

\*/

private case class ReplySource(

srcTweet: Tweet,

srcUser: User,

retweet: Option[Tweet] = None,

rtUser: Option[User] = None) {

private val photoTaggedUsers: Seq[User] =

srcTweet.mediaTags

.map(\_.tagMap.values.flatten)

.getOrElse(Nil)

.map(toUser)

.toSeq

private def toUser(mt: MediaTag): User =

mt match {

case MediaTag(\_, Some(id), Some(screenName), \_) => User(id, screenName)

case \_ => throw InvalidUserException

}

private def toUser(e: MentionEntity): User =

e match {

case MentionEntity(\_, \_, screenName, Some(id), \_, \_) => User(id, screenName)

case \_ => throw InvalidUserException

}

private def toUser(d: DirectedAtUser) = User(d.userId, d.screenName)

def allCardUsers(authorUser: User, cardUsersFinder: CardUsersFinder.Type): Future[Set[UserId]] =

Stitch.run(

cardUsersFinder(

CardUsersFinder.Request(

cardReference = getCardReference(srcTweet),

urls = getUrls(srcTweet).map(\_.url),

perspectiveUserId = authorUser.id

)

)

)

def srcTweetMentionedUsers: Seq[User] = getMentions(srcTweet).map(toUser)

private trait ReplyType {

val allExcludedUserIds: Set[UserId]

def directedAt: Option[User]

def requiredTextMention: Option[User]

def isExcluded(u: User): Boolean = allExcludedUserIds.contains(u.id)

def buildPrefix(otherMentions: Seq[User], maxImplicits: Int): String = {

val seen = new mutable.HashSet[UserId]

seen ++= allExcludedUserIds

// Never exclude the required mention

seen --= requiredTextMention.map(\_.id)

(requiredTextMention.toSeq ++ otherMentions)

.filter(u => seen.add(u.id))

.take(maxImplicits.max(requiredTextMention.size))

.map(u => s"@${u.screenName}")

.mkString(" ")

}

}

private case class SelfReply(

allExcludedUserIds: Set[UserId],

enableTweetToNarrowcasting: Boolean)

extends ReplyType {

private def srcTweetDirectedAt: Option[User] = getDirectedAtUser(srcTweet).map(toUser)

override def directedAt: Option[User] =

if (!enableTweetToNarrowcasting) None

else Seq.concat(rtUser, srcTweetDirectedAt).find(!isExcluded(\_))

override def requiredTextMention: Option[User] =

// Make sure the directedAt user is in the text to avoid confusion

directedAt

}

private case class BatchSubsequentReply(allExcludedUserIds: Set[UserId]) extends ReplyType {

override def directedAt: Option[User] = None

override def requiredTextMention: Option[User] = None

override def buildPrefix(otherMentions: Seq[User], maxImplicits: Int): String = ""

}

private case class RegularReply(

allExcludedUserIds: Set[UserId],

enableTweetToNarrowcasting: Boolean)

extends ReplyType {

override def directedAt: Option[User] =

Some(srcUser)

.filterNot(isExcluded)

.filter(\_ => enableTweetToNarrowcasting)

override def requiredTextMention: Option[User] =

// Include the source tweet's author as a mention in the reply, even if the reply is not

// narrowcasted to that user. All non-self-reply tweets require this mention.

Some(srcUser)

}

/\*\*

\* Computes an implicit mention prefix to add to the tweet text as well as any directed-at user.

\*

\* The first implicit mention is the source-tweet's author unless the reply is a self-reply, in

\* which case it inherits the DirectedAtUser from the source tweet, though the current author is

\* never added. This mention, if it exists, is the only mention that may be used to direct-at a

\* user and is the user that ends up in DirectedAtUserMetadata. If the user replied to a

\* retweet and the reply doesn't explicitly mention the retweet author, then the retweet author

\* will be next, followed by source tweet mentions and source tweet photo-tagged users.

\*

\* Users in excludedScreenNames originate from the PostTweetRequest and are filtered out of any

\* non-leading mention.

\*

\* Note on maxImplicits:

\* This method returns at most 'maxImplicits' mentions unless 'maxImplicits' is 0 and a

\* directed-at mention is required, in which case it returns 1. If this happens the reply may

\* fail downstream validation checks (e.g. TweetBuilder). With 280 visible character limit it's

\* theoretically possible to explicitly mention 93 users (280 / 3) but this bug shouldn't really

\* be an issue because:

\* 1.) Most replies don't have 50 explicit mentions

\* 2.) TOO-clients have switched to batchMode=Subsequent for self-replies which disable

source tweet's directed-at user inheritance

\* 3.) Requests rarely are rejected due to mention\_limit\_exceeded

\* If this becomes a problem we could reopen the mention limit discussion, specifically if the

\* backend should allow 51 while the explicit limit remains at 50.

\*

\* Note on batchMode:

\* Implicit mention prefix will be empty string if batchMode is BatchSubsequent. This is to

\* support batch composer.

\*/

def implicitMentionPrefixAndDAU(

maxImplicits: Int,

excludedUsers: Seq[User],

author: User,

enableTweetToNarrowcasting: Boolean,

batchMode: Option[BatchComposeMode]

): (String, Option[User]) = {

def allExcludedUserIds =

(excludedUsers ++ Seq(author)).map(\_.id).toSet

val replyType =

if (author.id == srcUser.id) {

if (batchMode.contains(BatchComposeMode.BatchSubsequent)) {

BatchSubsequentReply(allExcludedUserIds)

} else {

SelfReply(allExcludedUserIds, enableTweetToNarrowcasting)

}

} else {

RegularReply(allExcludedUserIds, enableTweetToNarrowcasting)

}

val prefix =

replyType.buildPrefix(

otherMentions = List.concat(rtUser, srcTweetMentionedUsers, photoTaggedUsers),

maxImplicits = maxImplicits

)

(prefix, replyType.directedAt)

}

/\*\*

\* Finds the longest possible prefix of whitespace separated @-mentions, restricted to

\* @-mentions that are derived from the reply chain.

\*/

def hideablePrefix(

text: String,

cardUsers: Seq[User],

explicitMentions: Seq[Extractor.Entity]

): Offset.CodePoint = {

val allowedMentions =

(srcTweetMentionedUsers.toSet + srcUser ++ rtUser.toSet ++ photoTaggedUsers ++ cardUsers)

.map(\_.screenName.toLowerCase)

val len = Offset.CodeUnit.length(text)

// To allow NO-BREAK SPACE' (U+00A0) in the prefix need .isSpaceChar

def isWhitespace(c: Char) = c.isWhitespace || c.isSpaceChar

@tailrec

def skipWs(offset: Offset.CodeUnit): Offset.CodeUnit =

if (offset == len || !isWhitespace(text.charAt(offset.toInt))) offset

else skipWs(offset.incr)

@tailrec

def go(offset: Offset.CodeUnit, mentions: Stream[Extractor.Entity]): Offset.CodeUnit =

if (offset == len) offset

else {

mentions match {

// if we are at the next mention, and it is allowed, skip past and recurse

case next #:: tail if next.getStart == offset.toInt =>

if (!allowedMentions.contains(next.getValue.toLowerCase)) offset

else go(skipWs(Offset.CodeUnit(next.getEnd)), tail)

// we found non-mention text

case \_ => offset

}

}

go(Offset.CodeUnit(0), explicitMentions.toStream).toCodePoint(text)

}

}

private def replyToUser(user: User, inReplyToStatusId: Option[TweetId] = None): Reply =

Reply(

inReplyToUserId = user.id,

inReplyToScreenName = Some(user.screenName),

inReplyToStatusId = inReplyToStatusId

)

/\*\*

\* A builder that generates reply from `inReplyToTweetId` or tweet text

\*

\* There are two kinds of "reply":

\* 1. reply to tweet, which is generated from `inReplyToTweetId`.

\*

\* A valid reply-to-tweet satisfies the following conditions:

\* 1). the tweet that is in-reply-to exists (and is visible to the user creating the tweet)

\* 2). the author of the in-reply-to tweet is mentioned anywhere in the tweet, or

\* this is a tweet that is in reply to the author's own tweet

\*

\* 2. reply to user, is generated when the tweet text starts with @user\_name. This is only

\* attempted if PostTweetRequest.enableTweetToNarrowcasting is true (default).

\*/

def apply(

userIdentityRepo: UserIdentityRepository.Type,

tweetRepo: TweetRepository.Optional,

replyCardUsersFinder: CardUsersFinder.Type,

selfThreadBuilder: SelfThreadBuilder,

relationshipRepo: RelationshipRepository.Type,

unmentionedEntitiesRepo: UnmentionedEntitiesRepository.Type,

enableRemoveUnmentionedImplicits: Gate[Unit],

stats: StatsReceiver,

maxMentions: Int

): Type = {

val exceptionCounters = ExceptionCounter(stats)

val modeScope = stats.scope("mode")

val compatModeCounter = modeScope.counter("compat")

val simpleModeCounter = modeScope.counter("simple")

def getUser(key: UserKey): Future[Option[User]] =

Stitch.run(

userIdentityRepo(key)

.map(ident => User(ident.id, ident.screenName))

.liftNotFoundToOption

)

def getUsers(userIds: Seq[UserId]): Future[Seq[ReplyBuilder.User]] =

Stitch.run(

Stitch

.traverse(userIds)(id => userIdentityRepo(UserKey(id)).liftNotFoundToOption)

.map(\_.flatten)

.map { identities => identities.map { ident => User(ident.id, ident.screenName) } }

)

val tweetQueryIncludes =

TweetQuery.Include(

tweetFields = Set(

Tweet.CoreDataField.id,

Tweet.CardReferenceField.id,

Tweet.CommunitiesField.id,

Tweet.MediaTagsField.id,

Tweet.MentionsField.id,

Tweet.UrlsField.id,

Tweet.EditControlField.id

) ++ selfThreadBuilder.requiredReplySourceFields.map(\_.id)

)

def tweetQueryOptions(forUserId: UserId) =

TweetQuery.Options(

tweetQueryIncludes,

forUserId = Some(forUserId),

enforceVisibilityFiltering = true

)

def getTweet(tweetId: TweetId, forUserId: UserId): Future[Option[Tweet]] =

Stitch.run(tweetRepo(tweetId, tweetQueryOptions(forUserId)))

def checkBlockRelationship(authorId: UserId, result: Result): Future[Unit] = {

val inReplyToBlocksTweeter =

RelationshipKey.blocks(

sourceId = result.reply.inReplyToUserId,

destinationId = authorId

)

Stitch.run(relationshipRepo(inReplyToBlocksTweeter)).flatMap {

case true => Future.exception(InReplyToTweetNotFound)

case false => Future.Unit

}

}

def checkIPIPolicy(request: Request, reply: Reply): Future[Unit] = {

if (request.spamResult == Spam.DisabledByIpiPolicy) {

Future.exception(Spam.DisabledByIpiFailure(reply.inReplyToScreenName))

} else {

Future.Unit

}

}

def getUnmentionedUsers(replySource: ReplySource): Future[Seq[UserId]] = {

if (enableRemoveUnmentionedImplicits()) {

val srcDirectedAt = replySource.srcTweet.directedAtUserMetadata.flatMap(\_.userId)

val srcTweetMentions = replySource.srcTweet.mentions.getOrElse(Nil).flatMap(\_.userId)

val idsToCheck = srcTweetMentions ++ srcDirectedAt

val conversationId = replySource.srcTweet.coreData.flatMap(\_.conversationId)

conversationId match {

case Some(cid) if idsToCheck.nonEmpty =>

stats.counter("unmentioned\_implicits\_check").incr()

Stitch

.run(unmentionedEntitiesRepo(cid, idsToCheck)).liftToTry.map {

case Return(Some(unmentionedUserIds)) =>

unmentionedUserIds

case \_ => Seq[UserId]()

}

case \_ => Future.Nil

}

} else {

Future.Nil

}

}

/\*\*

\* Constructs a `ReplySource` for the given `tweetId`, which captures the source tweet to be

\* replied to, its author, and if `tweetId` is for a retweet of the source tweet, then also

\* that retweet and its author. If the source tweet (or a retweet of it), or a corresponding

\* author, can't be found or isn't visible to the replier, then `InReplyToTweetNotFound` is

\* thrown.

\*/

def getReplySource(tweetId: TweetId, forUserId: UserId): Future[ReplySource] =

for {

tweet <- getTweet(tweetId, forUserId).flatMap {

case None => Future.exception(InReplyToTweetNotFound)

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

}

user <- getUser(UserKey(getUserId(tweet))).flatMap {

case None => Future.exception(InReplyToTweetNotFound)

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

}

res <- getShare(tweet) match {

case None => Future.value(ReplySource(tweet, user))

case Some(share) =>

// if the user is replying to a retweet, find the retweet source tweet,

// then update with the retweet and author.

getReplySource(share.sourceStatusId, forUserId)

.map(\_.copy(retweet = Some(tweet), rtUser = Some(user)))

}

} yield res

/\*\*

\* Computes a `Result` for the reply-to-tweet case. If `inReplyToTweetId` is for a retweet,

\* the reply will be computed against the source tweet. If `prependImplicitMentions` is true

\* and source tweet can't be found or isn't visible to replier, then this method will return

\* a `InReplyToTweetNotFound` failure. If `prependImplicitMentions` is false, then the reply

\* text must either mention the source tweet user, or it must be a reply to self; if both of

\* those conditions fail, then `None` is returned.

\*/

def makeReplyToTweet(

inReplyToTweetId: TweetId,

text: String,

author: User,

prependImplicitMentions: Boolean,

enableTweetToNarrowcasting: Boolean,

excludeUserIds: Seq[UserId],

batchMode: Option[BatchComposeMode]

): Future[Option[Result]] = {

val explicitMentions: Seq[Extractor.Entity] =

extractor.extractMentionedScreennamesWithIndices(text).asScala.toSeq

val mentionedScreenNames =

explicitMentions.map(\_.getValue.toLowerCase).toSet

/\*\*

\* If `prependImplicitMentions` is true, or the reply author is the same as the in-reply-to

\* author, then the reply text doesn't have to mention the in-reply-to author. Otherwise,

\* check that the text contains a mention of the reply author.

\*/

def isValidReplyTo(inReplyToUser: User): Boolean =

prependImplicitMentions ||

(inReplyToUser.id == author.id) ||

mentionedScreenNames.contains(inReplyToUser.screenName.toLowerCase)

getReplySource(inReplyToTweetId, author.id)

.flatMap { replySrc =>

val baseResult = BaseResult(

reply = replyToUser(replySrc.srcUser, Some(replySrc.srcTweet.id)),

conversationId = getConversationId(replySrc.srcTweet),

selfThreadMetadata = selfThreadBuilder.build(author.id, replySrc.srcTweet),

community = replySrc.srcTweet.communities,

// Reply tweets retain the same exclusive

// tweet controls as the tweet being replied to.

exclusiveTweetControl = replySrc.srcTweet.exclusiveTweetControl,

trustedFriendsControl = replySrc.srcTweet.trustedFriendsControl,

editControl = replySrc.srcTweet.editControl

)

if (isValidReplyTo(replySrc.srcUser)) {

if (prependImplicitMentions) {

// Simplified Replies mode - append server-side generated prefix to passed in text

simpleModeCounter.incr()

// remove the in-reply-to tweet author from the excluded users, in-reply-to tweet author will always be a directedAtUser

val filteredExcludedIds =

excludeUserIds.filterNot(uid => uid == TweetLenses.userId(replySrc.srcTweet))

for {

unmentionedUserIds <- getUnmentionedUsers(replySrc)

excludedUsers <- getUsers(filteredExcludedIds ++ unmentionedUserIds)

(prefix, directedAtUser) = replySrc.implicitMentionPrefixAndDAU(

maxImplicits = math.max(0, maxMentions - explicitMentions.size),

excludedUsers = excludedUsers,

author = author,

enableTweetToNarrowcasting = enableTweetToNarrowcasting,

batchMode = batchMode

)

} yield {

// prefix or text (or both) can be empty strings. Add " " separator and adjust

// prefix length only when both prefix and text are non-empty.

val textChunks = Seq(prefix, text).map(\_.trim).filter(\_.nonEmpty)

val tweetText = textChunks.mkString(" ")

val visibleStart =

if (textChunks.size == 2) {

Offset.CodePoint.length(prefix + " ")

} else {

Offset.CodePoint.length(prefix)

}

Some(

baseResult.toResult(

tweetText = tweetText,

directedAtMetadata = DirectedAtUserMetadata(directedAtUser.map(\_.id)),

visibleStart = visibleStart

)

)

}

} else {

// Backwards-compatibility mode - walk from beginning of text until find visibleStart

compatModeCounter.incr()

for {

cardUserIds <- replySrc.allCardUsers(author, replyCardUsersFinder)

cardUsers <- getUsers(cardUserIds.toSeq)

optUserIdentity <- extractReplyToUser(text)

directedAtUserId = optUserIdentity.map(\_.id).filter(\_ => enableTweetToNarrowcasting)

} yield {

Some(

baseResult.toResult(

tweetText = text,

directedAtMetadata = DirectedAtUserMetadata(directedAtUserId),

visibleStart = replySrc.hideablePrefix(text, cardUsers, explicitMentions),

)

)

}

}

} else {

Future.None

}

}

.handle {

// if `getReplySource` throws this exception, but we aren't computing implicit

// mentions, then we fall back to the reply-to-user case instead of reply-to-tweet

case InReplyToTweetNotFound if !prependImplicitMentions => None

}

}

def makeReplyToUser(text: String): Future[Option[Result]] =

extractReplyToUser(text).map(\_.map { user =>

Result(replyToUser(user), text, DirectedAtUserMetadata(Some(user.id)))

})

def extractReplyToUser(text: String): Future[Option[User]] =

Option(extractor.extractReplyScreenname(text)) match {

case None => Future.None

case Some(screenName) => getUser(UserKey(screenName))

}

FutureArrow[Request, Option[Result]] { request =>

exceptionCounters {

(request.inReplyToTweetId.filter(\_ > 0) match {

case None =>

Future.None

case Some(tweetId) =>

makeReplyToTweet(

tweetId,

request.tweetText,

User(request.authorId, request.authorScreenName),

request.prependImplicitMentions,

request.enableTweetToNarrowcasting,

request.excludeUserIds,

request.batchMode

)

}).flatMap {

case Some(r) =>

// Ensure that the author of this reply is not blocked by

// the user who they are replying to.

checkBlockRelationship(request.authorId, r)

.before(checkIPIPolicy(request, r.reply))

.before(Future.value(Some(r)))

case None if request.enableTweetToNarrowcasting =>

// We don't check the block relationship when the tweet is

// not part of a conversation (which is to say, we allow

// directed-at tweets from a blocked user.) These tweets

// will not cause notifications for the blocking user,

// despite the presence of the reply struct.

makeReplyToUser(request.tweetText)

case None =>

Future.None

}

}

}

}

}