package com.twitter.tweetypie

package handler

import com.twitter.featureswitches.v2.FeatureSwitchResults

import com.twitter.featureswitches.v2.FeatureSwitches

import com.twitter.gizmoduck.thriftscala.AccessPolicy

import com.twitter.gizmoduck.thriftscala.LabelValue

import com.twitter.gizmoduck.thriftscala.UserType

import com.twitter.snowflake.id.SnowflakeId

import com.twitter.stitch.NotFound

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.additionalfields.AdditionalFields.\_

import com.twitter.tweetypie.core.\_

import com.twitter.tweetypie.jiminy.tweetypie.NudgeBuilder

import com.twitter.tweetypie.jiminy.tweetypie.NudgeBuilderRequest

import com.twitter.tweetypie.media.Media

import com.twitter.tweetypie.repository.StratoCommunityAccessRepository.CommunityAccess

import com.twitter.tweetypie.repository.\_

import com.twitter.tweetypie.serverutil.DeviceSourceParser

import com.twitter.tweetypie.serverutil.ExtendedTweetMetadataBuilder

import com.twitter.tweetypie.store.\_

import com.twitter.tweetypie.thriftscala.\_

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

import com.twitter.tweetypie.tweettext.\_

import com.twitter.tweetypie.util.CommunityAnnotation

import com.twitter.tweetypie.util.CommunityUtil

import com.twitter.twittertext.Regex.{VALID\_URL => UrlPattern}

import com.twitter.twittertext.TwitterTextParser

case class TweetBuilderResult(

tweet: Tweet,

user: User,

createdAt: Time,

sourceTweet: Option[Tweet] = None,

sourceUser: Option[User] = None,

parentUserId: Option[UserId] = None,

isSilentFail: Boolean = false,

geoSearchRequestId: Option[GeoSearchRequestId] = None,

initialTweetUpdateRequest: Option[InitialTweetUpdateRequest] = None)

object TweetBuilder {

import GizmoduckUserCountsUpdatingStore.isUserTweet

import PostTweet.\_

import Preprocessor.\_

import TweetCreateState.{Spam => CreateStateSpam, \_}

import TweetText.\_

import UpstreamFailure.\_

type Type = FutureArrow[PostTweetRequest, TweetBuilderResult]

val log: Logger = Logger(getClass)

private[this] val \_unitMutation = Future.value(Mutation.unit[Any])

def MutationUnitFuture[T]: Future[Mutation[T]] = \_unitMutation.asInstanceOf[Future[Mutation[T]]]

case class MissingConversationId(inReplyToTweetId: TweetId) extends RuntimeException

case class TextVisibility(

visibleTextRange: Option[TextRange],

totalTextDisplayLength: Offset.DisplayUnit,

visibleText: String) {

val isExtendedTweet: Boolean = totalTextDisplayLength.toInt > OriginalMaxDisplayLength

/\*\*

\* Going forward we will be moving away from quoted-tweets urls in tweet text, but we

\* have a backwards-compat layer in Tweetypie which adds the QT url to text to provide

\* support for all clients to read in a backwards-compatible way until they upgrade.

\*

\* Tweets can become extended as their display length can go beyond 140

\* after adding the QT short url. Therefore, we are adding below function

\* to account for legacy formatting during read-time and generate a self-permalink.

\*/

def isExtendedWithExtraChars(extraChars: Int): Boolean =

totalTextDisplayLength.toInt > (OriginalMaxDisplayLength - extraChars)

}

/\*\* Max number of users that can be tagged on a single tweet \*/

val MaxMediaTagCount = 10

val MobileWebApp = "oauth:49152"

val M2App = "oauth:3033294"

val M5App = "oauth:3033300"

val TestRateLimitUserRole = "stresstest"

/\*\*

\* The fields to fetch for the user creating the tweet.

\*/

val userFields: Set[UserField] =

Set(

UserField.Profile,

UserField.ProfileDesign,

UserField.Account,

UserField.Safety,

UserField.Counts,

UserField.Roles,

UserField.UrlEntities,

UserField.Labels

)

/\*\*

\* The fields to fetch for the user of the source tweet in a retweet.

\*/

val sourceUserFields: Set[UserField] =

userFields + UserField.View

/\*\*

\* Converts repository exceptions into an API-compatible exception type

\*/

def convertRepoExceptions[A](

notFoundState: TweetCreateState,

failureHandler: Throwable => Throwable

): PartialFunction[Throwable, Stitch[A]] = {

// stitch.NotFound is converted to the supplied TweetCreateState, wrapped in TweetCreateFailure

case NotFound => Stitch.exception(TweetCreateFailure.State(notFoundState))

// OverCapacity exceptions should not be translated and should bubble up to the top

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

// Other exceptions are wrapped in the supplied failureHandler

case ex => Stitch.exception(failureHandler(ex))

}

/\*\*

\* Adapts a UserRepository to a Repository for looking up a single user and that

\* fails with an appropriate TweetCreateFailure if the user is not found.

\*/

def userLookup(userRepo: UserRepository.Type): UserId => Stitch[User] = {

val opts = UserQueryOptions(queryFields = userFields, visibility = UserVisibility.All)

userId =>

userRepo(UserKey(userId), opts)

.rescue(convertRepoExceptions[User](UserNotFound, UserLookupFailure(\_)))

}

/\*\*

\* Adapts a UserRepository to a Repository for looking up a single user and that

\* fails with an appropriate TweetCreateFailure if the user is not found.

\*/

def sourceUserLookup(userRepo: UserRepository.Type): (UserId, UserId) => Stitch[User] = {

val opts = UserQueryOptions(queryFields = sourceUserFields, visibility = UserVisibility.All)

(userId, forUserId) =>

userRepo(UserKey(userId), opts.copy(forUserId = Some(forUserId)))

.rescue(convertRepoExceptions[User](SourceUserNotFound, UserLookupFailure(\_)))

}

/\*\*

\* Any fields that are loaded on the user via TweetBuilder/RetweetBuilder, but which should not

\* be included on the user in the async-insert actions (such as hosebird) should be removed here.

\*

\* This will include perspectival fields that were loaded relative to the user creating the tweet.

\*/

def scrubUserInAsyncInserts: User => User =

user => user.copy(view = None)

/\*\*

\* Any fields that are loaded on the source user via TweetBuilder/RetweetBuilder, but which

\* should not be included on the user in the async-insert actions (such as hosebird) should

\* be removed here.

\*

\* This will include perspectival fields that were loaded relative to the user creating the tweet.

\*/

def scrubSourceUserInAsyncInserts: User => User =

// currently the same as scrubUserInAsyncInserts, could be different in the future

scrubUserInAsyncInserts

/\*\*

\* Any fields that are loaded on the source tweet via RetweetBuilder, but which should not be

\* included on the source tweetypie in the async-insert actions (such as hosebird) should

\* be removed here.

\*

\* This will include perspectival fields that were loaded relative to the user creating the tweet.

\*/

def scrubSourceTweetInAsyncInserts: Tweet => Tweet =

tweet => tweet.copy(perspective = None, cards = None, card2 = None)

/\*\*

\* Adapts a DeviceSource to a Repository for looking up a single device-source and that

\* fails with an appropriate TweetCreateFailure if not found.

\*/

def deviceSourceLookup(devSrcRepo: DeviceSourceRepository.Type): DeviceSourceRepository.Type =

appIdStr => {

val result: Stitch[DeviceSource] =

if (DeviceSourceParser.isValid(appIdStr)) {

devSrcRepo(appIdStr)

} else {

Stitch.exception(NotFound)

}

result.rescue(convertRepoExceptions(DeviceSourceNotFound, DeviceSourceLookupFailure(\_)))

}

/\*\*

\* Checks:

\* - that we have all the user fields we need

\* - that the user is active

\* - that they are not a frictionless follower account

\*/

def validateUser(user: User): Future[Unit] =

if (user.safety.isEmpty)

Future.exception(UserSafetyEmptyException)

else if (user.profile.isEmpty)

Future.exception(UserProfileEmptyException)

else if (user.safety.get.deactivated)

Future.exception(TweetCreateFailure.State(UserDeactivated))

else if (user.safety.get.suspended)

Future.exception(TweetCreateFailure.State(UserSuspended))

else if (user.labels.exists(\_.labels.exists(\_.labelValue == LabelValue.ReadOnly)))

Future.exception(TweetCreateFailure.State(CreateStateSpam))

else if (user.userType == UserType.Frictionless)

Future.exception(TweetCreateFailure.State(UserNotFound))

else if (user.userType == UserType.Soft)

Future.exception(TweetCreateFailure.State(UserNotFound))

else if (user.safety.get.accessPolicy == AccessPolicy.BounceAll ||

user.safety.get.accessPolicy == AccessPolicy.BounceAllPublicWrites)

Future.exception(TweetCreateFailure.State(UserReadonly))

else

Future.Unit

def validateCommunityReply(

communities: Option[Communities],

replyResult: Option[ReplyBuilder.Result]

): Future[Unit] = {

if (replyResult.flatMap(\_.reply.inReplyToStatusId).nonEmpty) {

val rootCommunities = replyResult.flatMap(\_.community)

val rootCommunityIds = CommunityUtil.communityIds(rootCommunities)

val replyCommunityIds = CommunityUtil.communityIds(communities)

if (rootCommunityIds == replyCommunityIds) {

Future.Unit

} else {

Future.exception(TweetCreateFailure.State(CommunityReplyTweetNotAllowed))

}

} else {

Future.Unit

}

}

// Project requirements do not allow exclusive tweets to be replies.

// All exclusive tweets must be root tweets.

def validateExclusiveTweetNotReplies(

exclusiveTweetControls: Option[ExclusiveTweetControl],

replyResult: Option[ReplyBuilder.Result]

): Future[Unit] = {

val isInReplyToTweet = replyResult.exists(\_.reply.inReplyToStatusId.isDefined)

if (exclusiveTweetControls.isDefined && isInReplyToTweet) {

Future.exception(TweetCreateFailure.State(SuperFollowsInvalidParams))

} else {

Future.Unit

}

}

// Invalid parameters for Exclusive Tweets:

// - Community field set # Tweets can not be both at the same time.

def validateExclusiveTweetParams(

exclusiveTweetControls: Option[ExclusiveTweetControl],

communities: Option[Communities]

): Future[Unit] = {

if (exclusiveTweetControls.isDefined && CommunityUtil.hasCommunity(communities)) {

Future.exception(TweetCreateFailure.State(SuperFollowsInvalidParams))

} else {

Future.Unit

}

}

def validateTrustedFriendsNotReplies(

trustedFriendsControl: Option[TrustedFriendsControl],

replyResult: Option[ReplyBuilder.Result]

): Future[Unit] = {

val isInReplyToTweet = replyResult.exists(\_.reply.inReplyToStatusId.isDefined)

if (trustedFriendsControl.isDefined && isInReplyToTweet) {

Future.exception(TweetCreateFailure.State(TrustedFriendsInvalidParams))

} else {

Future.Unit

}

}

def validateTrustedFriendsParams(

trustedFriendsControl: Option[TrustedFriendsControl],

conversationControl: Option[TweetCreateConversationControl],

communities: Option[Communities],

exclusiveTweetControl: Option[ExclusiveTweetControl]

): Future[Unit] = {

if (trustedFriendsControl.isDefined &&

(conversationControl.isDefined || CommunityUtil.hasCommunity(

communities) || exclusiveTweetControl.isDefined)) {

Future.exception(TweetCreateFailure.State(TrustedFriendsInvalidParams))

} else {

Future.Unit

}

}

/\*\*

\* Checks the weighted tweet text length using twitter-text, as used by clients.

\* This should ensure that any tweet the client deems valid will also be deemed

\* valid by Tweetypie.

\*/

def prevalidateTextLength(text: String, stats: StatsReceiver): Future[Unit] = {

val twitterTextConfig = TwitterTextParser.TWITTER\_TEXT\_DEFAULT\_CONFIG

val twitterTextResult = TwitterTextParser.parseTweet(text, twitterTextConfig)

val textTooLong = !twitterTextResult.isValid && text.length > 0

Future.when(textTooLong) {

val weightedLength = twitterTextResult.weightedLength

log.debug(

s"Weighted length too long. weightedLength: $weightedLength" +

s", Tweet text: '${diffshow.show(text)}'"

)

stats.counter("check\_weighted\_length/text\_too\_long").incr()

Future.exception(TweetCreateFailure.State(TextTooLong))

}

}

/\*\*

\* Checks that the tweet text is neither blank nor too long.

\*/

def validateTextLength(

text: String,

visibleText: String,

replyResult: Option[ReplyBuilder.Result],

stats: StatsReceiver

): Future[Unit] = {

val utf8Length = Offset.Utf8.length(text)

def visibleTextTooLong =

Offset.DisplayUnit.length(visibleText) > Offset.DisplayUnit(MaxVisibleWeightedEmojiLength)

def utf8LengthTooLong =

utf8Length > Offset.Utf8(MaxUtf8Length)

if (isBlank(text)) {

stats.counter("validate\_text\_length/text\_cannot\_be\_blank").incr()

Future.exception(TweetCreateFailure.State(TextCannotBeBlank))

} else if (replyResult.exists(\_.replyTextIsEmpty(text))) {

stats.counter("validate\_text\_length/reply\_text\_cannot\_be\_blank").incr()

Future.exception(TweetCreateFailure.State(TextCannotBeBlank))

} else if (visibleTextTooLong) {

// Final check that visible text does not exceed MaxVisibleWeightedEmojiLength

// characters.

// prevalidateTextLength() does some portion of validation as well, most notably

// weighted length on raw, unescaped text.

stats.counter("validate\_text\_length/text\_too\_long.visible\_length\_explicit").incr()

log.debug(

s"Explicit MaxVisibleWeightedLength visible length check failed. " +

s"visibleText: '${diffshow.show(visibleText)}' and " +

s"total text: '${diffshow.show(text)}'"

)

Future.exception(TweetCreateFailure.State(TextTooLong))

} else if (utf8LengthTooLong) {

stats.counter("validate\_text\_length/text\_too\_long.utf8\_length").incr()

Future.exception(TweetCreateFailure.State(TextTooLong))

} else {

stats.stat("validate\_text\_length/utf8\_length").add(utf8Length.toInt)

Future.Unit

}

}

def getTextVisibility(

text: String,

replyResult: Option[ReplyBuilder.Result],

urlEntities: Seq[UrlEntity],

mediaEntities: Seq[MediaEntity],

attachmentUrl: Option[String]

): TextVisibility = {

val totalTextLength = Offset.CodePoint.length(text)

val totalTextDisplayLength = Offset.DisplayUnit.length(text)

/\*\*

\* visibleEnd for multiple scenarios:

\*

\* normal tweet + media - fromIndex of mediaEntity (hydrated from last media permalink)

\* quote tweet + media - fromIndex of mediaEntity

\* replies + media - fromIndex of mediaEntity

\* normal quote tweet - total text length (visible text range will be None)

\* tweets with other attachments (DM deep links)

\* fromIndex of the last URL entity

\*/

val visibleEnd = mediaEntities.headOption

.map(\_.fromIndex)

.orElse(attachmentUrl.flatMap(\_ => urlEntities.lastOption).map(\_.fromIndex))

.map(from => (from - 1).max(0)) // for whitespace, unless there is none

.map(Offset.CodePoint(\_))

.getOrElse(totalTextLength)

val visibleStart = replyResult match {

case Some(rr) => rr.visibleStart.min(visibleEnd)

case None => Offset.CodePoint(0)

}

if (visibleStart.toInt == 0 && visibleEnd == totalTextLength) {

TextVisibility(

visibleTextRange = None,

totalTextDisplayLength = totalTextDisplayLength,

visibleText = text

)

} else {

val charFrom = visibleStart.toCodeUnit(text)

val charTo = charFrom.offsetByCodePoints(text, visibleEnd - visibleStart)

val visibleText = text.substring(charFrom.toInt, charTo.toInt)

TextVisibility(

visibleTextRange = Some(TextRange(visibleStart.toInt, visibleEnd.toInt)),

totalTextDisplayLength = totalTextDisplayLength,

visibleText = visibleText

)

}

}

def isValidHashtag(entity: HashtagEntity): Boolean =

TweetText.codePointLength(entity.text) <= TweetText.MaxHashtagLength

/\*\*

\* Validates that the number of various entities are within the limits, and the

\* length of hashtags are with the limit.

\*/

def validateEntities(tweet: Tweet): Future[Unit] =

if (getMentions(tweet).length > TweetText.MaxMentions)

Future.exception(TweetCreateFailure.State(MentionLimitExceeded))

else if (getUrls(tweet).length > TweetText.MaxUrls)

Future.exception(TweetCreateFailure.State(UrlLimitExceeded))

else if (getHashtags(tweet).length > TweetText.MaxHashtags)

Future.exception(TweetCreateFailure.State(HashtagLimitExceeded))

else if (getCashtags(tweet).length > TweetText.MaxCashtags)

Future.exception(TweetCreateFailure.State(CashtagLimitExceeded))

else if (getHashtags(tweet).exists(e => !isValidHashtag(e)))

Future.exception(TweetCreateFailure.State(HashtagLengthLimitExceeded))

else

Future.Unit

/\*\*

\* Update the user to what it should look like after the tweet is created

\*/

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

(user: User, tweet: Tweet) => {

val countAsUserTweet = isUserTweet(tweet)

val tweetsDelta = if (countAsUserTweet) 1 else 0

val mediaTweetsDelta = if (countAsUserTweet && hasMedia(tweet)) 1 else 0

Future.value(

user.copy(

counts = user.counts.map { counts =>

counts.copy(

tweets = counts.tweets + tweetsDelta,

mediaTweets = counts.mediaTweets.map(\_ + mediaTweetsDelta)

)

}

)

)

}

def validateAdditionalFields[R](implicit view: RequestView[R]): FutureEffect[R] =

FutureEffect[R] { req =>

view

.additionalFields(req)

.map(tweet =>

unsettableAdditionalFieldIds(tweet) ++ rejectedAdditionalFieldIds(tweet)) match {

case Some(unsettableFieldIds) if unsettableFieldIds.nonEmpty =>

Future.exception(

TweetCreateFailure.State(

InvalidAdditionalField,

Some(unsettableAdditionalFieldIdsErrorMessage(unsettableFieldIds))

)

)

case \_ => Future.Unit

}

}

def validateTweetMediaTags(

stats: StatsReceiver,

getUserMediaTagRateLimit: RateLimitChecker.GetRemaining,

userRepo: UserRepository.Optional

): (Tweet, Boolean) => Future[Mutation[Tweet]] = {

val userRepoWithStats: UserRepository.Optional =

(userKey, queryOptions) =>

userRepo(userKey, queryOptions).liftToTry.map {

case Return(res @ Some(\_)) =>

stats.counter("found").incr()

res

case Return(None) =>

stats.counter("not\_found").incr()

None

case Throw(\_) =>

stats.counter("failed").incr()

None

}

(tweet: Tweet, dark: Boolean) => {

val mediaTags = getMediaTagMap(tweet)

if (mediaTags.isEmpty) {

MutationUnitFuture

} else {

getUserMediaTagRateLimit((getUserId(tweet), dark)).flatMap { remainingMediaTagCount =>

val maxMediaTagCount = math.min(remainingMediaTagCount, MaxMediaTagCount)

val taggedUserIds =

mediaTags.values.flatten.toSeq.collect {

case MediaTag(MediaTagType.User, Some(userId), \_, \_) => userId

}.distinct

val droppedTagCount = taggedUserIds.size - maxMediaTagCount

if (droppedTagCount > 0) stats.counter("over\_limit\_tags").incr(droppedTagCount)

val userQueryOpts =

UserQueryOptions(

queryFields = Set(UserField.MediaView),

visibility = UserVisibility.MediaTaggable,

forUserId = Some(getUserId(tweet))

)

val keys = taggedUserIds.take(maxMediaTagCount).map(UserKey.byId)

val keyOpts = keys.map((\_, userQueryOpts))

Stitch.run {

Stitch

.traverse(keyOpts)(userRepoWithStats.tupled)

.map(\_.flatten)

.map { users =>

val userMap = users.map(u => u.id -> u).toMap

val mediaTagsMutation =

Mutation[Seq[MediaTag]] { mediaTags =>

val validMediaTags =

mediaTags.filter {

case MediaTag(MediaTagType.User, Some(userId), \_, \_) =>

userMap.get(userId).exists(\_.mediaView.exists(\_.canMediaTag))

case \_ => false

}

val invalidCount = mediaTags.size - validMediaTags.size

if (invalidCount != 0) {

stats.counter("invalid").incr(invalidCount)

Some(validMediaTags)

} else {

None

}

}

TweetLenses.mediaTagMap.mutation(mediaTagsMutation.liftMapValues)

}

}

}

}

}

}

def validateCommunityMembership(

communityMembershipRepository: StratoCommunityMembershipRepository.Type,

communityAccessRepository: StratoCommunityAccessRepository.Type,

communities: Option[Communities]

): Future[Unit] =

communities match {

case Some(Communities(Seq(communityId))) =>

Stitch

.run {

communityMembershipRepository(communityId).flatMap {

case true => Stitch.value(None)

case false =>

communityAccessRepository(communityId).map {

case Some(CommunityAccess.Public) | Some(CommunityAccess.Closed) =>

Some(TweetCreateState.CommunityUserNotAuthorized)

case Some(CommunityAccess.Private) | None =>

Some(TweetCreateState.CommunityNotFound)

}

}

}.flatMap {

case None =>

Future.Done

case Some(tweetCreateState) =>

Future.exception(TweetCreateFailure.State(tweetCreateState))

}

case Some(Communities(communities)) if communities.length > 1 =>

// Not allowed to specify more than one community ID.

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

case \_ => Future.Done

}

private[this] val CardUriSchemeRegex = "(?i)^(?:card|tombstone):".r

/\*\*

\* Is the given String a URI that is allowed as a card reference

\* without a matching URL in the text?

\*/

def hasCardsUriScheme(uri: String): Boolean =

CardUriSchemeRegex.findPrefixMatchOf(uri).isDefined

val InvalidAdditionalFieldEmptyUrlEntities: TweetCreateFailure.State =

TweetCreateFailure.State(

TweetCreateState.InvalidAdditionalField,

Some("url entities are empty")

)

val InvalidAdditionalFieldNonMatchingUrlAndShortUrl: TweetCreateFailure.State =

TweetCreateFailure.State(

TweetCreateState.InvalidAdditionalField,

Some("non-matching url and short url")

)

val InvalidAdditionalFieldInvalidUri: TweetCreateFailure.State =

TweetCreateFailure.State(

TweetCreateState.InvalidAdditionalField,

Some("invalid URI")

)

val InvalidAdditionalFieldInvalidCardUri: TweetCreateFailure.State =

TweetCreateFailure.State(

TweetCreateState.InvalidAdditionalField,

Some("invalid card URI")

)

type CardReferenceBuilder =

(Tweet, UrlShortener.Context) => Future[Mutation[Tweet]]

def cardReferenceBuilder(

cardReferenceValidator: CardReferenceValidationHandler.Type,

urlShortener: UrlShortener.Type

): CardReferenceBuilder =

(tweet, urlShortenerCtx) => {

getCardReference(tweet) match {

case Some(CardReference(uri)) =>

for {

cardUri <-

if (hasCardsUriScheme(uri)) {

// This is an explicit card references that does not

// need a corresponding URL in the text.

Future.value(uri)

} else if (UrlPattern.matcher(uri).matches) {

// The card reference is being used to specify which URL

// card to show. We need to verify that the URL is

// actually in the tweet text, or it can be effectively

// used to bypass the tweet length limit.

val urlEntities = getUrls(tweet)

if (urlEntities.isEmpty) {

// Fail fast if there can't possibly be a matching URL entity

Future.exception(InvalidAdditionalFieldEmptyUrlEntities)

} else {

// Look for the URL in the expanded URL entities. If

// it is present, then map it to the t.co shortened

// version of the URL.

urlEntities

.collectFirst {

case urlEntity if urlEntity.expanded.exists(\_ == uri) =>

Future.value(urlEntity.url)

}

.getOrElse {

// The URL may have been altered when it was

// returned from Talon, such as expanding a pasted

// t.co link. In this case, we t.co-ize the link and

// make sure that the corresponding t.co is present

// as a URL entity.

urlShortener((uri, urlShortenerCtx)).flatMap { shortened =>

if (urlEntities.exists(\_.url == shortened.shortUrl)) {

Future.value(shortened.shortUrl)

} else {

Future.exception(InvalidAdditionalFieldNonMatchingUrlAndShortUrl)

}

}

}

}

} else {

Future.exception(InvalidAdditionalFieldInvalidUri)

}

validatedCardUri <- cardReferenceValidator((getUserId(tweet), cardUri)).rescue {

case CardReferenceValidationFailedException =>

Future.exception(InvalidAdditionalFieldInvalidCardUri)

}

} yield {

TweetLenses.cardReference.mutation(

Mutation[CardReference] { cardReference =>

Some(cardReference.copy(cardUri = validatedCardUri))

}.checkEq.liftOption

)

}

case None =>

MutationUnitFuture

}

}

def filterInvalidData(

validateTweetMediaTags: (Tweet, Boolean) => Future[Mutation[Tweet]],

cardReferenceBuilder: CardReferenceBuilder

): (Tweet, PostTweetRequest, UrlShortener.Context) => Future[Tweet] =

(tweet: Tweet, request: PostTweetRequest, urlShortenerCtx: UrlShortener.Context) => {

Future

.join(

validateTweetMediaTags(tweet, request.dark),

cardReferenceBuilder(tweet, urlShortenerCtx)

)

.map {

case (mediaMutation, cardRefMutation) =>

mediaMutation.also(cardRefMutation).endo(tweet)

}

}

def apply(

stats: StatsReceiver,

validateRequest: PostTweetRequest => Future[Unit],

validateEdit: EditValidator.Type,

validateUser: User => Future[Unit] = TweetBuilder.validateUser,

validateUpdateRateLimit: RateLimitChecker.Validate,

tweetIdGenerator: TweetIdGenerator,

userRepo: UserRepository.Type,

deviceSourceRepo: DeviceSourceRepository.Type,

communityMembershipRepo: StratoCommunityMembershipRepository.Type,

communityAccessRepo: StratoCommunityAccessRepository.Type,

urlShortener: UrlShortener.Type,

urlEntityBuilder: UrlEntityBuilder.Type,

geoBuilder: GeoBuilder.Type,

replyBuilder: ReplyBuilder.Type,

mediaBuilder: MediaBuilder.Type,

attachmentBuilder: AttachmentBuilder.Type,

duplicateTweetFinder: DuplicateTweetFinder.Type,

spamChecker: Spam.Checker[TweetSpamRequest],

filterInvalidData: (Tweet, PostTweetRequest, UrlShortener.Context) => Future[Tweet],

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

validateConversationControl: ConversationControlBuilder.Validate.Type,

conversationControlBuilder: ConversationControlBuilder.Type,

validateTweetWrite: TweetWriteValidator.Type,

nudgeBuilder: NudgeBuilder.Type,

communitiesValidator: CommunitiesValidator.Type,

collabControlBuilder: CollabControlBuilder.Type,

editControlBuilder: EditControlBuilder.Type,

featureSwitches: FeatureSwitches

): TweetBuilder.Type = {

val entityExtractor = EntityExtractor.mutationWithoutUrls.endo

val getUser = userLookup(userRepo)

val getDeviceSource = deviceSourceLookup(deviceSourceRepo)

// create a tco of the permalink for given a tweetId

val permalinkShortener = (tweetId: TweetId, ctx: UrlShortener.Context) =>

urlShortener((s"https://twitter.com/i/web/status/$tweetId", ctx)).rescue {

// propagate OverCapacity

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

// convert any other failure into UrlShorteningFailure

case e => Future.exception(UrlShorteningFailure(e))

}

def extractGeoSearchRequestId(tweetGeoOpt: Option[TweetCreateGeo]): Option[GeoSearchRequestId] =

for {

tweetGeo <- tweetGeoOpt

geoSearchRequestId <- tweetGeo.geoSearchRequestId

} yield GeoSearchRequestId(geoSearchRequestId.id)

def featureSwitchResults(user: User, stats: StatsReceiver): Option[FeatureSwitchResults] =

TwitterContext()

.flatMap { viewer =>

UserViewerRecipient(user, viewer, stats)

}.map { recipient =>

featureSwitches.matchRecipient(recipient)

}

FutureArrow { request =>

for {

() <- validateRequest(request)

(tweetId, user, devsrc) <- Future.join(

tweetIdGenerator().rescue { case t => Future.exception(SnowflakeFailure(t)) },

Stitch.run(getUser(request.userId)),

Stitch.run(getDeviceSource(request.createdVia))

)

() <- validateUser(user)

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

// Feature Switch results are calculated once and shared between multiple builders

matchedResults = featureSwitchResults(user, stats)

() <- validateConversationControl(

ConversationControlBuilder.Validate.Request(

matchedResults = matchedResults,

conversationControl = request.conversationControl,

inReplyToTweetId = request.inReplyToTweetId

)

)

// strip illegal chars, normalize newlines, collapse blank lines, etc.

text = preprocessText(request.text)

() <- prevalidateTextLength(text, stats)

attachmentResult <- attachmentBuilder(

AttachmentBuilderRequest(

tweetId = tweetId,

user = user,

mediaUploadIds = request.mediaUploadIds,

cardReference = request.additionalFields.flatMap(\_.cardReference),

attachmentUrl = request.attachmentUrl,

remoteHost = request.remoteHost,

darkTraffic = request.dark,

deviceSource = devsrc

)

)

// updated text with appended attachment url, if any.

text <- Future.value(

attachmentResult.attachmentUrl match {

case None => text

case Some(url) => s"$text $url"

}

)

spamResult <- spamChecker(

TweetSpamRequest(

tweetId = tweetId,

userId = request.userId,

text = text,

mediaTags = request.additionalFields.flatMap(\_.mediaTags),

safetyMetaData = request.safetyMetaData,

inReplyToTweetId = request.inReplyToTweetId,

quotedTweetId = attachmentResult.quotedTweet.map(\_.tweetId),

quotedTweetUserId = attachmentResult.quotedTweet.map(\_.userId)

)

)

safety = user.safety.get

createdAt = SnowflakeId(tweetId).time

urlShortenerCtx = UrlShortener.Context(

tweetId = tweetId,

userId = user.id,

createdAt = createdAt,

userProtected = safety.isProtected,

clientAppId = devsrc.clientAppId,

remoteHost = request.remoteHost,

dark = request.dark

)

replyRequest = ReplyBuilder.Request(

authorId = request.userId,

authorScreenName = user.profile.map(\_.screenName).get,

inReplyToTweetId = request.inReplyToTweetId,

tweetText = text,

prependImplicitMentions = request.autoPopulateReplyMetadata,

enableTweetToNarrowcasting = request.enableTweetToNarrowcasting,

excludeUserIds = request.excludeReplyUserIds.getOrElse(Nil),

spamResult = spamResult,

batchMode = request.transientContext.flatMap(\_.batchCompose)

)

replyResult <- replyBuilder(replyRequest)

replyOpt = replyResult.map(\_.reply)

replyConversationId <- replyResult match {

case Some(r) if r.reply.inReplyToStatusId.nonEmpty =>

r.conversationId match {

case None =>

// Throw this specific exception to make it easier to

// count how often we hit this corner case.

Future.exception(MissingConversationId(r.reply.inReplyToStatusId.get))

case conversationIdOpt => Future.value(conversationIdOpt)

}

case \_ => Future.value(None)

}

// Validate that the current user can reply to this conversation, based on

// the conversation's ConversationControl.

// Note: currently we only validate conversation controls access on replies,

// therefore we use the conversationId from the inReplyToStatus.

// Validate that the exclusive tweet control option is only used by allowed users.

() <- validateTweetWrite(

TweetWriteValidator.Request(

replyConversationId,

request.userId,

request.exclusiveTweetControlOptions,

replyResult.flatMap(\_.exclusiveTweetControl),

request.trustedFriendsControlOptions,

replyResult.flatMap(\_.trustedFriendsControl),

attachmentResult.quotedTweet,

replyResult.flatMap(\_.reply.inReplyToStatusId),

replyResult.flatMap(\_.editControl),

request.editOptions

)

)

convoId = replyConversationId match {

case Some(replyConvoId) => replyConvoId

case None =>

// This is a root tweet, so the tweet id is the conversation id.

tweetId

}

() <- nudgeBuilder(

NudgeBuilderRequest(

text = text,

inReplyToTweetId = replyOpt.flatMap(\_.inReplyToStatusId),

conversationId = if (convoId == tweetId) None else Some(convoId),

hasQuotedTweet = attachmentResult.quotedTweet.nonEmpty,

nudgeOptions = request.nudgeOptions,

tweetId = Some(tweetId),

)

)

// updated text with implicit reply mentions inserted, if any

text <- Future.value(

replyResult.map(\_.tweetText).getOrElse(text)

)

// updated text with urls replaced with t.cos

((text, urlEntities), (geoCoords, placeIdOpt)) <- Future.join(

urlEntityBuilder((text, urlShortenerCtx))

.map {

case (text, urlEntities) =>

UrlEntityBuilder.updateTextAndUrls(text, urlEntities)(partialHtmlEncode)

},

if (request.geo.isEmpty)

Future.value((None, None))

else

geoBuilder(

GeoBuilder.Request(

request.geo.get,

user.account.map(\_.geoEnabled).getOrElse(false),

user.account.map(\_.language).getOrElse("en")

)

).map(r => (r.geoCoordinates, r.placeId))

)

// updated text with trailing media url

MediaBuilder.Result(text, mediaEntities, mediaKeys) <-

request.mediaUploadIds.getOrElse(Nil) match {

case Nil => Future.value(MediaBuilder.Result(text, Nil, Nil))

case ids =>

mediaBuilder(

MediaBuilder.Request(

mediaUploadIds = ids,

text = text,

tweetId = tweetId,

userId = user.id,

userScreenName = user.profile.get.screenName,

isProtected = user.safety.get.isProtected,

createdAt = createdAt,

dark = request.dark,

productMetadata = request.mediaMetadata.map(\_.toMap)

)

)

}

() <- Future.when(!request.dark) {

val reqInfo =

DuplicateTweetFinder.RequestInfo.fromPostTweetRequest(request, text)

duplicateTweetFinder(reqInfo).flatMap {

case None => Future.Unit

case Some(duplicateId) =>

log.debug(s"timeline\_duplicate\_check\_failed:$duplicateId")

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

}

}

textVisibility = getTextVisibility(

text = text,

replyResult = replyResult,

urlEntities = urlEntities,

mediaEntities = mediaEntities,

attachmentUrl = attachmentResult.attachmentUrl

)

() <- validateTextLength(

text = text,

visibleText = textVisibility.visibleText,

replyResult = replyResult,

stats = stats

)

communities =

request.additionalFields

.flatMap(CommunityAnnotation.additionalFieldsToCommunityIDs)

.map(ids => Communities(communityIds = ids))

rootExclusiveControls = request.exclusiveTweetControlOptions.map { \_ =>

ExclusiveTweetControl(request.userId)

}

() <- validateExclusiveTweetNotReplies(rootExclusiveControls, replyResult)

() <- validateExclusiveTweetParams(rootExclusiveControls, communities)

replyExclusiveControls = replyResult.flatMap(\_.exclusiveTweetControl)

// The userId is pulled off of the request rather than being supplied

// via the ExclusiveTweetControlOptions because additional fields

// can be set by clients to contain any value they want.

// This could include userIds that don't match their actual userId.

// Only one of replyResult or request.exclusiveTweetControlOptions will be defined.

exclusiveTweetControl = replyExclusiveControls.orElse(rootExclusiveControls)

rootTrustedFriendsControl = request.trustedFriendsControlOptions.map { options =>

TrustedFriendsControl(options.trustedFriendsListId)

}

() <- validateTrustedFriendsNotReplies(rootTrustedFriendsControl, replyResult)

() <- validateTrustedFriendsParams(

rootTrustedFriendsControl,

request.conversationControl,

communities,

exclusiveTweetControl

)

replyTrustedFriendsControl = replyResult.flatMap(\_.trustedFriendsControl)

trustedFriendsControl = replyTrustedFriendsControl.orElse(rootTrustedFriendsControl)

collabControl <- collabControlBuilder(

CollabControlBuilder.Request(

collabControlOptions = request.collabControlOptions,

replyResult = replyResult,

communities = communities,

trustedFriendsControl = trustedFriendsControl,

conversationControl = request.conversationControl,

exclusiveTweetControl = exclusiveTweetControl,

userId = request.userId

))

isCollabInvitation = collabControl.isDefined && (collabControl.get match {

case CollabControl.CollabInvitation(\_: CollabInvitation) => true

case \_ => false

})

coreData = TweetCoreData(

userId = request.userId,

text = text,

createdAtSecs = createdAt.inSeconds,

createdVia = devsrc.internalName,

reply = replyOpt,

hasTakedown = safety.hasTakedown,

// We want to nullcast community tweets and CollabInvitations

// This will disable tweet fanout to followers' home timelines,

// and filter the tweets from appearing from the tweeter's profile

// or search results for the tweeter's tweets.

nullcast =

request.nullcast || CommunityUtil.hasCommunity(communities) || isCollabInvitation,

narrowcast = request.narrowcast,

nsfwUser = request.possiblySensitive.getOrElse(safety.nsfwUser),

nsfwAdmin = safety.nsfwAdmin,

trackingId = request.trackingId,

placeId = placeIdOpt,

coordinates = geoCoords,

conversationId = Some(convoId),

// Set hasMedia to true if we know that there is media,

// and leave it unknown if not, so that it will be

// correctly set for pasted media.

hasMedia = if (mediaEntities.nonEmpty) Some(true) else None

)

tweet = Tweet(

id = tweetId,

coreData = Some(coreData),

urls = Some(urlEntities),

media = Some(mediaEntities),

mediaKeys = if (mediaKeys.nonEmpty) Some(mediaKeys) else None,

contributor = getContributor(request.userId),

visibleTextRange = textVisibility.visibleTextRange,

selfThreadMetadata = replyResult.flatMap(\_.selfThreadMetadata),

directedAtUserMetadata = replyResult.map(\_.directedAtMetadata),

composerSource = request.composerSource,

quotedTweet = attachmentResult.quotedTweet,

exclusiveTweetControl = exclusiveTweetControl,

trustedFriendsControl = trustedFriendsControl,

collabControl = collabControl,

noteTweet = request.noteTweetOptions.map(options =>

NoteTweet(options.noteTweetId, options.isExpandable))

)

editControl <- editControlBuilder(

EditControlBuilder.Request(

postTweetRequest = request,

tweet = tweet,

matchedResults = matchedResults

)

)

tweet <- Future.value(tweet.copy(editControl = editControl))

tweet <- Future.value(entityExtractor(tweet))

() <- validateEntities(tweet)

tweet <- {

val cctlRequest =

ConversationControlBuilder.Request.fromTweet(

tweet,

request.conversationControl,

request.noteTweetOptions.flatMap(\_.mentionedUserIds))

Stitch.run(conversationControlBuilder(cctlRequest)).map { conversationControl =>

tweet.copy(conversationControl = conversationControl)

}

}

tweet <- Future.value(

setAdditionalFields(tweet, request.additionalFields)

)

() <- validateCommunityMembership(communityMembershipRepo, communityAccessRepo, communities)

() <- validateCommunityReply(communities, replyResult)

() <- communitiesValidator(

CommunitiesValidator.Request(matchedResults, safety.isProtected, communities))

tweet <- Future.value(tweet.copy(communities = communities))

tweet <- Future.value(

tweet.copy(underlyingCreativesContainerId = request.underlyingCreativesContainerId)

)

// For certain tweets we want to write a self-permalink which is used to generate modified

// tweet text for legacy clients that contains a link. NOTE: this permalink is for

// the tweet being created - we also create permalinks for related tweets further down

// e.g. if this tweet is an edit, we might create a permalink for the initial tweet as well

tweet <- {

val isBeyond140 = textVisibility.isExtendedWithExtraChars(attachmentResult.extraChars)

val isEditTweet = request.editOptions.isDefined

val isMixedMedia = Media.isMixedMedia(mediaEntities)

val isNoteTweet = request.noteTweetOptions.isDefined

if (isBeyond140 || isEditTweet || isMixedMedia || isNoteTweet)

permalinkShortener(tweetId, urlShortenerCtx)

.map { selfPermalink =>

tweet.copy(

selfPermalink = Some(selfPermalink),

extendedTweetMetadata = Some(ExtendedTweetMetadataBuilder(tweet, selfPermalink))

)

}

else {

Future.value(tweet)

}

}

// When an edit tweet is created we have to update some information on the

// initial tweet, this object stores info about those updates for use

// in the tweet insert store.

// We update the editControl for each edit tweet and for the first edit tweet

// we update the self permalink.

initialTweetUpdateRequest: Option[InitialTweetUpdateRequest] <- editControl match {

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

// Identifies the first edit of an initial tweet

val isFirstEdit =

request.editOptions.map(\_.previousTweetId).contains(edit.initialTweetId)

// A potential permalink for this tweet being created's initial tweet

val selfPermalinkForInitial: Future[Option[ShortenedUrl]] =

if (isFirstEdit) {

// `tweet` is the first edit of an initial tweet, which means

// we need to write a self permalink. We create it here in

// TweetBuilder and pass it through to the tweet store to

// be written to the initial tweet.

permalinkShortener(edit.initialTweetId, urlShortenerCtx).map(Some(\_))

} else {

Future.value(None)

}

selfPermalinkForInitial.map { link =>

Some(

InitialTweetUpdateRequest(

initialTweetId = edit.initialTweetId,

editTweetId = tweet.id,

selfPermalink = link

))

}

// This is not an edit this is the initial tweet - so there are no initial

// tweet updates

case \_ => Future.value(None)

}

tweet <- filterInvalidData(tweet, request, urlShortenerCtx)

() <- validateEdit(tweet, request.editOptions)

user <- updateUserCounts(user, tweet)

} yield {

TweetBuilderResult(

tweet,

user,

createdAt,

isSilentFail = spamResult == Spam.SilentFail,

geoSearchRequestId = extractGeoSearchRequestId(request.geo),

initialTweetUpdateRequest = initialTweetUpdateRequest

)

}

}

}

}