package com.twitter.tweetypie

package federated.columns

import com.twitter.accounts.util.SafetyMetadataUtils

import com.twitter.ads.callback.thriftscala.EngagementRequest

import com.twitter.bouncer.thriftscala.{Bounce => BouncerBounce}

import com.twitter.escherbird.thriftscala.TweetEntityAnnotation

import com.twitter.geo.model.LatitudeLongitude

import com.twitter.stitch.Stitch

import com.twitter.strato.catalog.OpMetadata

import com.twitter.strato.config.AllOf

import com.twitter.strato.config.BouncerAccess

import com.twitter.strato.config.ContactInfo

import com.twitter.strato.config.Policy

import com.twitter.strato.data.Conv

import com.twitter.strato.data.Description.PlainText

import com.twitter.strato.data.Lifecycle.Production

import com.twitter.strato.fed.StratoFed

import com.twitter.strato.opcontext.OpContext

import com.twitter.strato.response.Err

import com.twitter.strato.thrift.ScroogeConv

import com.twitter.tweetypie.decider.overrides.TweetyPieDeciderOverrides

import com.twitter.tweetypie.federated.columns.ApiErrors.\_

import com.twitter.tweetypie.federated.columns.CreateTweetColumn.toCreateTweetErr

import com.twitter.tweetypie.federated.context.GetRequestContext

import com.twitter.tweetypie.federated.prefetcheddata.PrefetchedDataRequest

import com.twitter.tweetypie.federated.prefetcheddata.PrefetchedDataResponse

import com.twitter.tweetypie.federated.promotedcontent.TweetPromotedContentLogger

import com.twitter.tweetypie.federated.promotedcontent.TweetPromotedContentLogger.\_

import com.twitter.tweetypie.repository.UnmentionInfoRepository

import com.twitter.tweetypie.repository.VibeRepository

import com.twitter.tweetypie.thriftscala.TransientCreateContext

import com.twitter.tweetypie.thriftscala.TweetCreateContextKey

import com.twitter.tweetypie.thriftscala.TweetCreateState.\_

import com.twitter.tweetypie.thriftscala.{graphql => gql}

import com.twitter.tweetypie.util.CommunityAnnotation

import com.twitter.tweetypie.util.ConversationControls

import com.twitter.tweetypie.util.TransientContextUtil

import com.twitter.tweetypie.{thriftscala => thrift}

import com.twitter.util.Throwables

import com.twitter.weaverbird.common.{GetRequestContext => WGetRequestContext}

class CreateTweetColumn(

postTweet: thrift.PostTweetRequest => Future[thrift.PostTweetResult],

getRequestContext: GetRequestContext,

prefetchedDataRepository: PrefetchedDataRequest => Stitch[PrefetchedDataResponse],

unmentionInfoRepository: UnmentionInfoRepository.Type,

vibeRepository: VibeRepository.Type,

logTweetPromotedContent: TweetPromotedContentLogger.Type,

statsReceiver: StatsReceiver,

enableCommunityTweetCreatesDecider: Gate[Unit],

) extends StratoFed.Column(CreateTweetColumn.Path)

with StratoFed.Execute.StitchWithContext

with StratoFed.HandleDarkRequests {

override val policy: Policy = AllOf(

Seq(AccessPolicy.TweetMutationCommonAccessPolicies, BouncerAccess()))

// The underlying call to thriftTweetService.postRetweet is not idempotent

override val isIdempotent: Boolean = false

override type Arg = gql.CreateTweetRequest

override type Result = gql.CreateTweetResponseWithSubqueryPrefetchItems

override val argConv: Conv[Arg] = ScroogeConv.fromStruct

override val resultConv: Conv[Result] = ScroogeConv.fromStruct

override val contactInfo: ContactInfo = TweetypieContactInfo

override val metadata: OpMetadata =

OpMetadata(

Some(Production),

Some(

PlainText(

"""

Creates a tweet using the calling authenticated Twitter user as author.

NOTE, not all Tweet space fields are GraphQL queryable in the CreateTweet mutation response.

See http://go/missing-create-tweet-fields.

"""))

)

private val getWeaverbirdCtx = new WGetRequestContext()

override def execute(request: Arg, opContext: OpContext): Stitch[Result] = {

val ctx = getRequestContext(opContext)

// First, do any request parameter validation that can result in an error

// prior to calling into thriftTweetService.postTweet.

val safetyLevel = ctx.safetyLevel.getOrElse(throw SafetyLevelMissingErr)

val trackingId = request.engagementRequest match {

case Some(engagementRequest: EngagementRequest) if ctx.hasPrivilegePromotedTweetsInTimeline =>

TrackingId.parse(engagementRequest.impressionId, statsReceiver)

case Some(e: EngagementRequest) =>

throw ClientNotPrivilegedErr

case None =>

None

}

val deviceSource = ctx.deviceSource.getOrElse(throw GenericAccessDeniedErr)

if (request.nullcast && !ctx.hasPrivilegeNullcastingAccess) {

throw GenericAccessDeniedErr

}

val safetyMetadata = SafetyMetadataUtils.makeSafetyMetaData(

sessionHash = ctx.sessionHash,

knownDeviceToken = ctx.knownDeviceToken,

contributorId = ctx.contributorId

)

val cardReference: Option[thrift.CardReference] =

request.cardUri.filter(\_.nonEmpty).map(thrift.CardReference(\_))

val escherbirdEntityAnnotations: Option[thrift.EscherbirdEntityAnnotations] =

request.semanticAnnotationIds

.filter(\_.nonEmpty)

.map((seq: Seq[gql.TweetAnnotation]) => seq.map(parseTweetEntityAnnotation))

.map(thrift.EscherbirdEntityAnnotations(\_))

val mediaEntities = request.media.map(\_.mediaEntities)

val mediaUploadIds = mediaEntities.map(\_.map(\_.mediaId)).filter(\_.nonEmpty)

val mediaTags: Option[thrift.TweetMediaTags] = {

val mediaTagsAuthorized = !ctx.isContributorRequest

val tagMap: Map[MediaId, Seq[thrift.MediaTag]] =

mediaEntities

.getOrElse(Nil)

.filter(\_ => mediaTagsAuthorized)

.filter(\_.taggedUsers.nonEmpty)

.map(mediaEntity =>

mediaEntity.mediaId ->

mediaEntity.taggedUsers

.map(user\_id => thrift.MediaTag(thrift.MediaTagType.User, Some(user\_id))))

.toMap

Option(tagMap)

.filter(\_.nonEmpty)

.map(thrift.TweetMediaTags(\_))

}

// Can not have both conversation controls and communities defined for a tweet

// as they have conflicting permissions on who can reply to the tweet.

val communities = parseCommunityIds(escherbirdEntityAnnotations)

if (request.conversationControl.isDefined && communities.nonEmpty) {

throw CannotConvoControlAndCommunitiesErr

}

// Currently we do not support posting to multiple communities.

if (communities.length > 1) {

throw TooManyCommunitiesErr

}

// Kill switch for community tweets in case we need to disable them for app security.

if (communities.nonEmpty && !enableCommunityTweetCreatesDecider()) {

throw CommunityUserNotAuthorizedErr

}

// additionalFields is used to marshal multiple input params and

// should only be defined if one or more of those params are defined.

val additionalFields: Option[Tweet] =

cardReference

.orElse(escherbirdEntityAnnotations)

.orElse(mediaTags)

.map(\_ =>

thrift.Tweet(

0L,

cardReference = cardReference,

escherbirdEntityAnnotations = escherbirdEntityAnnotations,

mediaTags = mediaTags

))

val transientContext: Option[TransientCreateContext] =

parseTransientContext(

request.batchCompose,

request.periscope,

ctx.twitterUserId,

)

// PostTweetRequest.additionalContext is marked as deprecated in favor of .transientContext,

// but the REST API still supports it and it is still passed along through Tweetypie, and

// FanoutService and Notifications still depend on it.

val additionalContext: Option[Map[TweetCreateContextKey, String]] =

transientContext.map(TransientContextUtil.toAdditionalContext)

val thriftPostTweetRequest = thrift.PostTweetRequest(

userId = ctx.twitterUserId,

text = request.tweetText,

createdVia = deviceSource,

inReplyToTweetId = request.reply.map(\_.inReplyToTweetId),

geo = request.geo.flatMap(parseTweetCreateGeo),

autoPopulateReplyMetadata = request.reply.isDefined,

excludeReplyUserIds = request.reply.map(\_.excludeReplyUserIds).filter(\_.nonEmpty),

nullcast = request.nullcast,

// Send a dark request to Tweetypie if the dark\_request directive is set or

// if the Tweet is undo-able.

dark = ctx.isDarkRequest || request.undoOptions.exists(\_.isUndo),

hydrationOptions = Some(HydrationOptions.writePathHydrationOptions(ctx.cardsPlatformKey)),

remoteHost = ctx.remoteHost,

safetyMetaData = Some(safetyMetadata),

attachmentUrl = request.attachmentUrl,

mediaUploadIds = mediaUploadIds,

mediaMetadata = None,

transientContext = transientContext,

additionalContext = additionalContext,

conversationControl = request.conversationControl.map(parseTweetCreateConversationControl),

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

thrift.ExclusiveTweetControlOptions()

},

trustedFriendsControlOptions =

request.trustedFriendsControlOptions.map(parseTrustedFriendsControlOptions),

editOptions = request.editOptions.flatMap(\_.previousTweetId.map(thrift.EditOptions(\_))),

collabControlOptions = request.collabControlOptions.map(parseCollabControlOptions),

additionalFields = additionalFields,

trackingId = trackingId,

noteTweetOptions = request.noteTweetOptions.map(options =>

thrift.NoteTweetOptions(

options.noteTweetId,

options.mentionedScreenNames,

options.mentionedUserIds,

options.isExpandable))

)

val stitchPostTweet =

Stitch.callFuture {

TweetyPieDeciderOverrides.ConversationControlUseFeatureSwitchResults.On {

postTweet(thriftPostTweetRequest)

}

}

for {

engagement <- request.engagementRequest

if !request.reply.exists(\_.inReplyToTweetId == 0) // no op per go/rb/845242

engagementType = if (request.reply.isDefined) ReplyEngagement else TweetEngagement

} logTweetPromotedContent(engagement, engagementType, ctx.isDarkRequest)

stitchPostTweet.flatMap { result: thrift.PostTweetResult =>

result.state match {

case thrift.TweetCreateState.Ok =>

val unmentionSuccessCounter = statsReceiver.counter("unmention\_info\_success")

val unmentionFailuresCounter = statsReceiver.counter("unmention\_info\_failures")

val unmentionFailuresScope = statsReceiver.scope("unmention\_info\_failures")

val unmentionInfoStitch = result.tweet match {

case Some(tweet) =>

unmentionInfoRepository(tweet)

.onFailure { t =>

unmentionFailuresCounter.incr()

unmentionFailuresScope.counter(Throwables.mkString(t): \_\*).incr()

}

.onSuccess { \_ =>

unmentionSuccessCounter.incr()

}

.rescue {

case \_ =>

Stitch.None

}

case \_ =>

Stitch.None

}

val vibeSuccessCounter = statsReceiver.counter("vibe\_success")

val vibeFailuresCounter = statsReceiver.counter("vibe\_failures")

val vibeFailuresScope = statsReceiver.scope("vibe\_failures")

val vibeStitch = result.tweet match {

case Some(tweet) =>

vibeRepository(tweet)

.onSuccess { \_ =>

vibeSuccessCounter.incr()

}

.onFailure { t =>

vibeFailuresCounter.incr()

vibeFailuresScope.counter(Throwables.mkString(t): \_\*).incr()

}

.rescue {

case \_ =>

Stitch.None

}

case \_ =>

Stitch.None

}

Stitch

.join(unmentionInfoStitch, vibeStitch)

.liftToOption()

.flatMap { prefetchFields =>

val r = PrefetchedDataRequest(

tweet = result.tweet.get,

sourceTweet = result.sourceTweet,

quotedTweet = result.quotedTweet,

safetyLevel = safetyLevel,

unmentionInfo = prefetchFields.flatMap(params => params.\_1),

vibe = prefetchFields.flatMap(params => params.\_2),

requestContext = getWeaverbirdCtx()

)

prefetchedDataRepository(r)

.liftToOption()

.map((prefetchedData: Option[PrefetchedDataResponse]) => {

gql.CreateTweetResponseWithSubqueryPrefetchItems(

data = Some(gql.CreateTweetResponse(result.tweet.map(\_.id))),

subqueryPrefetchItems = prefetchedData.map(\_.value)

)

})

}

case errState =>

throw toCreateTweetErr(errState, result.bounce, result.failureReason)

}

}

}

private[this] def parseTweetCreateGeo(gqlGeo: gql.TweetGeo): Option[thrift.TweetCreateGeo] = {

val coordinates: Option[thrift.GeoCoordinates] =

gqlGeo.coordinates.map { coords =>

LatitudeLongitude.of(coords.latitude, coords.longitude) match {

case Return(latlon: LatitudeLongitude) =>

thrift.GeoCoordinates(

latitude = latlon.latitudeDegrees,

longitude = latlon.longitudeDegrees,

geoPrecision = latlon.precision,

display = coords.displayCoordinates

)

case Throw(\_) =>

throw InvalidCoordinatesErr

}

}

val geoSearchRequestId = gqlGeo.geoSearchRequestId.map { id =>

if (id.isEmpty) {

throw InvalidGeoSearchRequestIdErr

}

thrift.TweetGeoSearchRequestID(id)

}

if (coordinates.isEmpty && gqlGeo.placeId.isEmpty) {

None

} else {

Some(

thrift.TweetCreateGeo(

coordinates = coordinates,

placeId = gqlGeo.placeId,

geoSearchRequestId = geoSearchRequestId

))

}

}

private[this] def parseTweetCreateConversationControl(

gqlCC: gql.TweetConversationControl

): thrift.TweetCreateConversationControl =

gqlCC.mode match {

case gql.ConversationControlMode.ByInvitation =>

ConversationControls.Create.byInvitation()

case gql.ConversationControlMode.Community =>

ConversationControls.Create.community()

case gql.ConversationControlMode.EnumUnknownConversationControlMode(\_) =>

throw ConversationControlNotSupportedErr

}

private[this] def parseTweetEntityAnnotation(

gqlTweetAnnotation: gql.TweetAnnotation

): TweetEntityAnnotation =

TweetEntityAnnotation(

gqlTweetAnnotation.groupId,

gqlTweetAnnotation.domainId,

gqlTweetAnnotation.entityId

)

private[this] def parseCommunityIds(

escherbirdAnnotations: Option[thrift.EscherbirdEntityAnnotations]

): Seq[Long] =

escherbirdAnnotations

.map(\_.entityAnnotations).getOrElse(Nil)

.flatMap {

case CommunityAnnotation(id) => Seq(id)

case \_ => Nil

}

private[this] def parseBatchMode(

gqlBatchComposeMode: gql.BatchComposeMode

): thrift.BatchComposeMode = {

gqlBatchComposeMode match {

case gql.BatchComposeMode.BatchFirst =>

thrift.BatchComposeMode.BatchFirst

case gql.BatchComposeMode.BatchSubsequent =>

thrift.BatchComposeMode.BatchSubsequent

case gql.BatchComposeMode.EnumUnknownBatchComposeMode(\_) =>

throw InvalidBatchModeParameterErr

}

}

private[this] def parseTransientContext(

gqlBatchComposeMode: Option[gql.BatchComposeMode],

gqlPeriscope: Option[gql.TweetPeriscopeContext],

twitterUserId: UserId,

): Option[TransientCreateContext] = {

val batchComposeMode = gqlBatchComposeMode.map(parseBatchMode)

// Per c.t.fanoutservice.model.Tweet#deviceFollowType, isLive=None and Some(false) are

// equivalent and the creatorId is discarded in both cases.

val periscopeIsLive = gqlPeriscope.map(\_.isLive).filter(\_ == true)

val periscopeCreatorId = if (periscopeIsLive.isDefined) Some(twitterUserId) else None

if (batchComposeMode.isDefined || periscopeIsLive.isDefined) {

Some(

thrift.TransientCreateContext(

batchCompose = batchComposeMode,

periscopeIsLive = periscopeIsLive,

periscopeCreatorId = periscopeCreatorId

)

)

} else {

None

}

}

private[this] def parseTrustedFriendsControlOptions(

gqlTrustedFriendsControlOptions: gql.TrustedFriendsControlOptions

): thrift.TrustedFriendsControlOptions = {

thrift.TrustedFriendsControlOptions(

trustedFriendsListId = gqlTrustedFriendsControlOptions.trustedFriendsListId

)

}

private[this] def parseCollabControlOptions(

gqlCollabControlOptions: gql.CollabControlOptions

): thrift.CollabControlOptions = {

gqlCollabControlOptions.collabControlType match {

case gql.CollabControlType.CollabInvitation =>

thrift.CollabControlOptions.CollabInvitation(

thrift.CollabInvitationOptions(

collaboratorUserIds = gqlCollabControlOptions.collaboratorUserIds

)

)

case gql.CollabControlType.EnumUnknownCollabControlType(\_) =>

throw CollabTweetInvalidParamsErr

}

}

}

object CreateTweetColumn {

val Path = "tweetypie/createTweet.Tweet"

def toCreateTweetErr(

errState: thrift.TweetCreateState,

bounce: Option[BouncerBounce],

failureReason: Option[String]

): Err = errState match {

case TextCannotBeBlank =>

TweetCannotBeBlankErr

case TextTooLong =>

TweetTextTooLongErr

case Duplicate =>

DuplicateStatusErr

case MalwareUrl =>

MalwareTweetErr

case UserDeactivated | UserSuspended =>

// should not occur since this condition is caught by access policy filters

CurrentUserSuspendedErr

case RateLimitExceeded =>

RateLimitExceededErr

case UrlSpam =>

TweetUrlSpamErr

case Spam | UserReadonly =>

TweetSpammerErr

case SpamCaptcha =>

CaptchaChallengeErr

case SafetyRateLimitExceeded =>

SafetyRateLimitExceededErr

case Bounce if bounce.isDefined =>

accessDeniedByBouncerErr(bounce.get)

case MentionLimitExceeded =>

MentionLimitExceededErr

case UrlLimitExceeded =>

UrlLimitExceededErr

case HashtagLimitExceeded =>

HashtagLimitExceededErr

case CashtagLimitExceeded =>

CashtagLimitExceededErr

case HashtagLengthLimitExceeded =>

HashtagLengthLimitExceededErr

case TooManyAttachmentTypes =>

TooManyAttachmentTypesErr

case InvalidUrl =>

InvalidUrlErr

case DisabledByIpiPolicy =>

failureReason

.map(tweetEngagementLimitedErr)

.getOrElse(GenericTweetCreateErr)

case InvalidAdditionalField =>

failureReason

.map(invalidAdditionalFieldWithReasonErr)

.getOrElse(InvalidAdditionalFieldErr)

// InvalidImage has been deprecated by tweetypie. Use InvalidMedia instead.

case InvalidMedia | InvalidImage | MediaNotFound =>

invalidMediaErr(failureReason)

case InReplyToTweetNotFound =>

InReplyToTweetNotFoundErr

case InvalidAttachmentUrl =>

InvalidAttachmentUrlErr

case ConversationControlNotAllowed =>

ConversationControlNotAuthorizedErr

case InvalidConversationControl =>

ConversationControlInvalidErr

case ReplyTweetNotAllowed =>

ConversationControlReplyRestricted

case ExclusiveTweetEngagementNotAllowed =>

ExclusiveTweetEngagementNotAllowedErr

case CommunityReplyTweetNotAllowed =>

CommunityReplyTweetNotAllowedErr

case CommunityUserNotAuthorized =>

CommunityUserNotAuthorizedErr

case CommunityNotFound =>

CommunityNotFoundErr

case SuperFollowsInvalidParams =>

SuperFollowInvalidParamsErr

case SuperFollowsCreateNotAuthorized =>

SuperFollowCreateNotAuthorizedErr

case CommunityProtectedUserCannotTweet =>

CommunityProtectedUserCannotTweetErr

case TrustedFriendsInvalidParams =>

TrustedFriendsInvalidParamsErr

case TrustedFriendsEngagementNotAllowed =>

TrustedFriendsEngagementNotAllowedErr

case TrustedFriendsCreateNotAllowed =>

TrustedFriendsCreateNotAllowedErr

case TrustedFriendsQuoteTweetNotAllowed =>

TrustedFriendsQuoteTweetNotAllowedErr

case CollabTweetInvalidParams =>

CollabTweetInvalidParamsErr

case StaleTweetEngagementNotAllowed =>

StaleTweetEngagementNotAllowedErr

case StaleTweetQuoteTweetNotAllowed =>

StaleTweetQuoteTweetNotAllowedErr

case FieldEditNotAllowed =>

FieldEditNotAllowedErr

case NotEligibleForEdit =>

NotEligibleForEditErr

case \_ =>

GenericTweetCreateErr

}

}