package com.twitter.tweetypie

package handler

import com.twitter.context.thriftscala.FeatureContext

import com.twitter.tweetypie.backends.LimiterService

import com.twitter.tweetypie.core.\_

import com.twitter.tweetypie.serverutil.ExceptionCounter

import com.twitter.tweetypie.store.InsertTweet

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.tweetypie.util.TweetCreationLock.{Key => TweetCreationLockKey}

object PostTweet {

type Type[R] = FutureArrow[R, PostTweetResult]

/\*\*

\* A type-class to abstract over tweet creation requests.

\*/

trait RequestView[R] {

def isDark(req: R): Boolean

def sourceTweetId(req: R): Option[TweetId]

def options(req: R): Option[WritePathHydrationOptions]

def userId(req: R): UserId

def uniquenessId(req: R): Option[Long]

def returnSuccessOnDuplicate(req: R): Boolean

def returnDuplicateTweet(req: R): Boolean =

returnSuccessOnDuplicate(req) || uniquenessId(req).nonEmpty

def lockKey(req: R): TweetCreationLockKey

def geo(req: R): Option[TweetCreateGeo]

def featureContext(req: R): Option[FeatureContext]

def additionalContext(req: R): Option[collection.Map[TweetCreateContextKey, String]]

def transientContext(req: R): Option[TransientCreateContext]

def additionalFields(req: R): Option[Tweet]

def duplicateState: TweetCreateState

def scope: String

def isNullcast(req: R): Boolean

def creativesContainerId(req: R): Option[CreativesContainerId]

def noteTweetMentionedUserIds(req: R): Option[Seq[Long]]

}

/\*\*

\* An implementation of `RequestView` for `PostTweetRequest`.

\*/

implicit object PostTweetRequestView extends RequestView[PostTweetRequest] {

def isDark(req: PostTweetRequest): Boolean = req.dark

def sourceTweetId(req: PostTweetRequest): None.type = None

def options(req: PostTweetRequest): Option[WritePathHydrationOptions] = req.hydrationOptions

def userId(req: PostTweetRequest): UserId = req.userId

def uniquenessId(req: PostTweetRequest): Option[Long] = req.uniquenessId

def returnSuccessOnDuplicate(req: PostTweetRequest) = false

def lockKey(req: PostTweetRequest): TweetCreationLockKey = TweetCreationLockKey.byRequest(req)

def geo(req: PostTweetRequest): Option[TweetCreateGeo] = req.geo

def featureContext(req: PostTweetRequest): Option[FeatureContext] = req.featureContext

def additionalContext(

req: PostTweetRequest

): Option[collection.Map[TweetCreateContextKey, String]] = req.additionalContext

def transientContext(req: PostTweetRequest): Option[TransientCreateContext] =

req.transientContext

def additionalFields(req: PostTweetRequest): Option[Tweet] = req.additionalFields

def duplicateState: TweetCreateState.Duplicate.type = TweetCreateState.Duplicate

def scope = "tweet"

def isNullcast(req: PostTweetRequest): Boolean = req.nullcast

def creativesContainerId(req: PostTweetRequest): Option[CreativesContainerId] =

req.underlyingCreativesContainerId

def noteTweetMentionedUserIds(req: PostTweetRequest): Option[Seq[Long]] =

req.noteTweetOptions match {

case Some(noteTweetOptions) => noteTweetOptions.mentionedUserIds

case \_ => None

}

}

/\*\*

\* An implementation of `RequestView` for `RetweetRequest`.

\*/

implicit object RetweetRequestView extends RequestView[RetweetRequest] {

def isDark(req: RetweetRequest): Boolean = req.dark

def sourceTweetId(req: RetweetRequest): None.type = None

def options(req: RetweetRequest): Option[WritePathHydrationOptions] = req.hydrationOptions

def userId(req: RetweetRequest): UserId = req.userId

def uniquenessId(req: RetweetRequest): Option[Long] = req.uniquenessId

def returnSuccessOnDuplicate(req: RetweetRequest): Boolean = req.returnSuccessOnDuplicate

def lockKey(req: RetweetRequest): TweetCreationLockKey =

req.uniquenessId match {

case Some(id) => TweetCreationLockKey.byUniquenessId(req.userId, id)

case None => TweetCreationLockKey.bySourceTweetId(req.userId, req.sourceStatusId)

}

def geo(req: RetweetRequest): None.type = None

def featureContext(req: RetweetRequest): Option[FeatureContext] = req.featureContext

def additionalContext(req: RetweetRequest): None.type = None

def transientContext(req: RetweetRequest): None.type = None

def additionalFields(req: RetweetRequest): Option[Tweet] = req.additionalFields

def duplicateState: TweetCreateState.AlreadyRetweeted.type = TweetCreateState.AlreadyRetweeted

def scope = "retweet"

def isNullcast(req: RetweetRequest): Boolean = req.nullcast

def creativesContainerId(req: RetweetRequest): Option[CreativesContainerId] = None

def noteTweetMentionedUserIds(req: RetweetRequest): Option[Seq[Long]] = None

}

/\*\*

\* A `Filter` is used to decorate a `FutureArrow` that has a known return type

\* and an input type for which there is a `RequestView` type-class instance.

\*/

trait Filter[Res] { self =>

type T[Req] = FutureArrow[Req, Res]

/\*\*

\* Wraps a base arrow with additional behavior.

\*/

def apply[Req: RequestView](base: T[Req]): T[Req]

/\*\*

\* Composes two filter. The resulting filter itself composes FutureArrows.

\*/

def andThen(next: Filter[Res]): Filter[Res] =

new Filter[Res] {

def apply[Req: RequestView](base: T[Req]): T[Req] =

next(self(base))

}

}

/\*\*

\* This filter attempts to prevent some race-condition related duplicate tweet creations,

\* via use of a `TweetCreateLock`. When a duplicate is detected, this filter can synthesize

\* a successful `PostTweetResult` if applicable, or return the appropriate coded response.

\*/

object DuplicateHandler {

def apply(

tweetCreationLock: TweetCreationLock,

getTweets: GetTweetsHandler.Type,

stats: StatsReceiver

): Filter[PostTweetResult] =

new Filter[PostTweetResult] {

def apply[R: RequestView](base: T[R]): T[R] = {

val view = implicitly[RequestView[R]]

val notFoundCount = stats.counter(view.scope, "not\_found")

val foundCounter = stats.counter(view.scope, "found")

FutureArrow.rec[R, PostTweetResult] { self => req =>

val duplicateKey = view.lockKey(req)

// attempts to find the duplicate tweet.

//

// if `returnDupTweet` is true and we find the tweet, then we return a

// successful `PostTweetResult` with that tweet. if we don't find the

// tweet, we throw an `InternalServerError`.

//

// if `returnDupTweet` is false and we find the tweet, then we return

// the appropriate duplicate state. if we don't find the tweet, then

// we unlock the duplicate key and try again.

def duplicate(tweetId: TweetId, returnDupTweet: Boolean) =

findDuplicate(tweetId, req).flatMap {

case Some(postTweetResult) =>

foundCounter.incr()

if (returnDupTweet) Future.value(postTweetResult)

else Future.value(PostTweetResult(state = view.duplicateState))

case None =>

notFoundCount.incr()

if (returnDupTweet) {

// If we failed to load the tweet, but we know that it

// should exist, then return an InternalServerError, so that

// the client treats it as a failed tweet creation req.

Future.exception(

InternalServerError("Failed to load duplicate existing tweet: " + tweetId)

)

} else {

// Assume the lock is stale if we can't load the tweet. It's

// possible that the lock is not stale, but the tweet is not

// yet available, which requires that it not be present in

// cache and not yet available from the backend. This means

// that the failure mode is to allow tweeting if we can't

// determine the state, but it should be rare that we can't

// determine it.

tweetCreationLock.unlock(duplicateKey).before(self(req))

}

}

tweetCreationLock(duplicateKey, view.isDark(req), view.isNullcast(req)) {

base(req)

}.rescue {

case TweetCreationInProgress =>

Future.value(PostTweetResult(state = TweetCreateState.Duplicate))

// if tweetCreationLock detected a duplicate, look up the duplicate

// and return the appropriate result

case DuplicateTweetCreation(tweetId) =>

duplicate(tweetId, view.returnDuplicateTweet(req))

// it's possible that tweetCreationLock didn't find a duplicate for a

// retweet attempt, but `RetweetBuilder` did.

case TweetCreateFailure.AlreadyRetweeted(tweetId) if view.returnDuplicateTweet(req) =>

duplicate(tweetId, true)

}

}

}

private def findDuplicate[R: RequestView](

tweetId: TweetId,

req: R

): Future[Option[PostTweetResult]] = {

val view = implicitly[RequestView[R]]

val readRequest =

GetTweetsRequest(

tweetIds = Seq(tweetId),

// Assume that the defaults are OK for all of the hydration

// options except the ones that are explicitly set in the

// req.

options = Some(

GetTweetOptions(

forUserId = Some(view.userId(req)),

includePerspectivals = true,

includeCards = view.options(req).exists(\_.includeCards),

cardsPlatformKey = view.options(req).flatMap(\_.cardsPlatformKey)

)

)

)

getTweets(readRequest).map {

case Seq(result) =>

if (result.tweetState == StatusState.Found) {

// If the tweet was successfully found, then convert the

// read result into a successful write result.

Some(

PostTweetResult(

TweetCreateState.Ok,

result.tweet,

// if the retweet is really old, the retweet perspective might no longer

// be available, but we want to maintain the invariant that the `postRetweet`

// endpoint always returns a source tweet with the correct perspective.

result.sourceTweet.map { srcTweet =>

TweetLenses.perspective

.update(\_.map(\_.copy(retweeted = true, retweetId = Some(tweetId))))

.apply(srcTweet)

},

result.quotedTweet

)

)

} else {

None

}

}

}

}

}

/\*\*

\* A `Filter` that applies rate limiting to failing requests.

\*/

object RateLimitFailures {

def apply(

validateLimit: RateLimitChecker.Validate,

incrementSuccess: LimiterService.IncrementByOne,

incrementFailure: LimiterService.IncrementByOne

): Filter[TweetBuilderResult] =

new Filter[TweetBuilderResult] {

def apply[R: RequestView](base: T[R]): T[R] = {

val view = implicitly[RequestView[R]]

FutureArrow[R, TweetBuilderResult] { req =>

val userId = view.userId(req)

val dark = view.isDark(req)

val contributorUserId: Option[UserId] = getContributor(userId).map(\_.userId)

validateLimit((userId, dark))

.before {

base(req).onFailure { \_ =>

// We don't increment the failure rate limit if the failure

// was from the failure rate limit so that the user can't

// get in a loop where tweet creation is never attempted. We

// don't increment it if the creation is dark because there

// is no way to perform a dark tweet creation through the

// API, so it's most likey some kind of test traffic like

// tap-compare.

if (!dark) incrementFailure(userId, contributorUserId)

}

}

.onSuccess { resp =>

// If we return a silent failure, then we want to

// increment the rate limit as if the tweet was fully

// created, because we want it to appear that way to the

// user whose creation silently failed.

if (resp.isSilentFail) incrementSuccess(userId, contributorUserId)

}

}

}

}

}

/\*\*

\* A `Filter` for counting non-`TweetCreateFailure` failures.

\*/

object CountFailures {

def apply[Res](stats: StatsReceiver, scopeSuffix: String = "\_builder"): Filter[Res] =

new Filter[Res] {

def apply[R: RequestView](base: T[R]): T[R] = {

val view = implicitly[RequestView[R]]

val exceptionCounter = ExceptionCounter(stats.scope(view.scope + scopeSuffix))

base.onFailure {

case (\_, \_: TweetCreateFailure) =>

case (\_, ex) => exceptionCounter(ex)

}

}

}

}

/\*\*

\* A `Filter` for logging failures.

\*/

object LogFailures extends Filter[PostTweetResult] {

private[this] val failedTweetCreationsLogger = Logger(

"com.twitter.tweetypie.FailedTweetCreations"

)

def apply[R: RequestView](base: T[R]): T[R] =

FutureArrow[R, PostTweetResult] { req =>

base(req).onFailure {

case failure => failedTweetCreationsLogger.info(s"request: $req\nfailure: $failure")

}

}

}

/\*\*

\* A `Filter` for converting a thrown `TweetCreateFailure` into a `PostTweetResult`.

\*/

object RescueTweetCreateFailure extends Filter[PostTweetResult] {

def apply[R: RequestView](base: T[R]): T[R] =

FutureArrow[R, PostTweetResult] { req =>

base(req).rescue {

case failure: TweetCreateFailure => Future.value(failure.toPostTweetResult)

}

}

}

/\*\*

\* Builds a base handler for `PostTweetRequest` and `RetweetRequest`. The handler

\* calls an underlying tweet builder, creates a `InsertTweet.Event`, hydrates

\* that, passes it to `tweetStore`, and then converts it to a `PostTweetResult`.

\*/

object Handler {

def apply[R: RequestView](

tweetBuilder: FutureArrow[R, TweetBuilderResult],

hydrateInsertEvent: FutureArrow[InsertTweet.Event, InsertTweet.Event],

tweetStore: InsertTweet.Store,

): Type[R] = {

FutureArrow { req =>

for {

bldrRes <- tweetBuilder(req)

event <- hydrateInsertEvent(toInsertTweetEvent(req, bldrRes))

\_ <- Future.when(!event.dark)(tweetStore.insertTweet(event))

} yield toPostTweetResult(event)

}

}

/\*\*

\* Converts a request/`TweetBuilderResult` pair into an `InsertTweet.Event`.

\*/

def toInsertTweetEvent[R: RequestView](

req: R,

bldrRes: TweetBuilderResult

): InsertTweet.Event = {

val view = implicitly[RequestView[R]]

InsertTweet.Event(

tweet = bldrRes.tweet,

user = bldrRes.user,

sourceTweet = bldrRes.sourceTweet,

sourceUser = bldrRes.sourceUser,

parentUserId = bldrRes.parentUserId,

timestamp = bldrRes.createdAt,

dark = view.isDark(req) || bldrRes.isSilentFail,

hydrateOptions = view.options(req).getOrElse(WritePathHydrationOptions()),

featureContext = view.featureContext(req),

initialTweetUpdateRequest = bldrRes.initialTweetUpdateRequest,

geoSearchRequestId = for {

geo <- view.geo(req)

searchRequestID <- geo.geoSearchRequestId

} yield {

GeoSearchRequestId(requestID = searchRequestID.id)

},

additionalContext = view.additionalContext(req),

transientContext = view.transientContext(req),

noteTweetMentionedUserIds = view.noteTweetMentionedUserIds(req)

)

}

/\*\*

\* Converts an `InsertTweet.Event` into a successful `PostTweetResult`.

\*/

def toPostTweetResult(event: InsertTweet.Event): PostTweetResult =

PostTweetResult(

TweetCreateState.Ok,

Some(event.tweet),

sourceTweet = event.sourceTweet,

quotedTweet = event.quotedTweet

)

}

}