package com.twitter.tweetypie

package repository

import com.twitter.snowflake.id.SnowflakeId

import com.twitter.stitch.NotFound

import com.twitter.stitch.Stitch

import com.twitter.tweetypie

import com.twitter.tweetypie.client\_id.ClientIdHelper

import com.twitter.tweetypie.core.\_

import com.twitter.tweetypie.storage.TweetStorageClient.GetStoredTweet

import com.twitter.tweetypie.storage.TweetStorageClient.GetTweet

import com.twitter.tweetypie.storage.\_

import scala.util.control.NoStackTrace

case class StorageGetTweetFailure(tweetId: TweetId, underlying: Throwable)

extends Exception(s"tweetId=$tweetId", underlying)

with NoStackTrace

object ManhattanTweetRepository {

private[this] val logger = Logger(getClass)

def apply(

getTweet: TweetStorageClient.GetTweet,

getStoredTweet: TweetStorageClient.GetStoredTweet,

shortCircuitLikelyPartialTweetReads: Gate[Duration],

statsReceiver: StatsReceiver,

clientIdHelper: ClientIdHelper,

): TweetResultRepository.Type = {

def likelyAvailable(tweetId: TweetId): Boolean =

if (SnowflakeId.isSnowflakeId(tweetId)) {

val tweetAge: Duration = Time.now.since(SnowflakeId(tweetId).time)

!shortCircuitLikelyPartialTweetReads(tweetAge)

} else {

true // Not a snowflake id, so should definitely be available

}

val likelyPartialTweetReadsCounter = statsReceiver.counter("likely\_partial\_tweet\_reads")

(tweetId, options) =>

if (!likelyAvailable(tweetId)) {

likelyPartialTweetReadsCounter.incr()

val currentClient =

clientIdHelper.effectiveClientId.getOrElse(ClientIdHelper.UnknownClientId)

logger.debug(s"likely\_partial\_tweet\_read $tweetId $currentClient")

Stitch.exception(NotFound)

} else if (options.fetchStoredTweets) {

getStoredTweet(tweetId).liftToTry.flatMap(handleGetStoredTweetResponse(tweetId, \_))

} else {

getTweet(tweetId).liftToTry.flatMap(handleGetTweetResponse(tweetId, \_))

}

}

private def handleGetTweetResponse(

tweetId: tweetypie.TweetId,

response: Try[GetTweet.Response]

): Stitch[TweetResult] = {

response match {

case Return(GetTweet.Response.Found(tweet)) =>

Stitch.value(TweetResult(TweetData(tweet = tweet), HydrationState.modified))

case Return(GetTweet.Response.NotFound) =>

Stitch.exception(NotFound)

case Return(GetTweet.Response.Deleted) =>

Stitch.exception(FilteredState.Unavailable.TweetDeleted)

case Return(\_: GetTweet.Response.BounceDeleted) =>

Stitch.exception(FilteredState.Unavailable.BounceDeleted)

case Throw(\_: storage.RateLimited) =>

Stitch.exception(OverCapacity(s"Storage overcapacity, tweetId=$tweetId"))

case Throw(e) =>

Stitch.exception(StorageGetTweetFailure(tweetId, e))

}

}

private def handleGetStoredTweetResponse(

tweetId: tweetypie.TweetId,

response: Try[GetStoredTweet.Response]

): Stitch[TweetResult] = {

def translateErrors(

getStoredTweetErrs: Seq[GetStoredTweet.Error]

): Seq[StoredTweetResult.Error] = {

getStoredTweetErrs.map {

case GetStoredTweet.Error.TweetIsCorrupt => StoredTweetResult.Error.Corrupt

case GetStoredTweet.Error.ScrubbedFieldsPresent =>

StoredTweetResult.Error.ScrubbedFieldsPresent

case GetStoredTweet.Error.TweetFieldsMissingOrInvalid =>

StoredTweetResult.Error.FieldsMissingOrInvalid

case GetStoredTweet.Error.TweetShouldBeHardDeleted =>

StoredTweetResult.Error.ShouldBeHardDeleted

}

}

def toTweetResult(

tweet: Tweet,

state: Option[TweetStateRecord],

errors: Seq[GetStoredTweet.Error]

): TweetResult = {

val translatedErrors = translateErrors(errors)

val canHydrate: Boolean =

!translatedErrors.contains(StoredTweetResult.Error.Corrupt) &&

!translatedErrors.contains(StoredTweetResult.Error.FieldsMissingOrInvalid)

val storedTweetResult = state match {

case None => StoredTweetResult.Present(translatedErrors, canHydrate)

case Some(TweetStateRecord.HardDeleted(\_, softDeletedAtMsec, hardDeletedAtMsec)) =>

StoredTweetResult.HardDeleted(softDeletedAtMsec, hardDeletedAtMsec)

case Some(TweetStateRecord.SoftDeleted(\_, softDeletedAtMsec)) =>

StoredTweetResult.SoftDeleted(softDeletedAtMsec, translatedErrors, canHydrate)

case Some(TweetStateRecord.BounceDeleted(\_, deletedAtMsec)) =>

StoredTweetResult.BounceDeleted(deletedAtMsec, translatedErrors, canHydrate)

case Some(TweetStateRecord.Undeleted(\_, undeletedAtMsec)) =>

StoredTweetResult.Undeleted(undeletedAtMsec, translatedErrors, canHydrate)

case Some(TweetStateRecord.ForceAdded(\_, addedAtMsec)) =>

StoredTweetResult.ForceAdded(addedAtMsec, translatedErrors, canHydrate)

}

TweetResult(

TweetData(tweet = tweet, storedTweetResult = Some(storedTweetResult)),

HydrationState.modified)

}

val tweetResult = response match {

case Return(GetStoredTweet.Response.FoundAny(tweet, state, \_, \_, errors)) =>

toTweetResult(tweet, state, errors)

case Return(GetStoredTweet.Response.Failed(tweetId, \_, \_, \_, errors)) =>

val tweetData = TweetData(

tweet = Tweet(tweetId),

storedTweetResult = Some(StoredTweetResult.Failed(translateErrors(errors))))

TweetResult(tweetData, HydrationState.modified)

case Return(GetStoredTweet.Response.HardDeleted(tweetId, state, \_, \_)) =>

toTweetResult(Tweet(tweetId), state, Seq())

case Return(GetStoredTweet.Response.NotFound(tweetId)) => {

val tweetData = TweetData(

tweet = Tweet(tweetId),

storedTweetResult = Some(StoredTweetResult.NotFound)

)

TweetResult(tweetData, HydrationState.modified)

}

case \_ => {

val tweetData = TweetData(

tweet = Tweet(tweetId),

storedTweetResult = Some(StoredTweetResult.Failed(Seq(StoredTweetResult.Error.Corrupt))))

TweetResult(tweetData, HydrationState.modified)

}

}

Stitch.value(tweetResult)

}

}