/\*\* Copyright 2010 Twitter, Inc. \*/

package com.twitter.tweetypie

package store

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.additionalfields.AdditionalFields

import com.twitter.tweetypie.storage.Field

import com.twitter.tweetypie.storage.Response.TweetResponse

import com.twitter.tweetypie.storage.Response.TweetResponseCode

import com.twitter.tweetypie.storage.TweetStorageClient

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

import com.twitter.tweetypie.storage.TweetStorageException

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.util.Future

case class UpdateTweetNotFoundException(tweetId: TweetId) extends Exception

trait ManhattanTweetStore

extends TweetStoreBase[ManhattanTweetStore]

with InsertTweet.Store

with AsyncDeleteTweet.Store

with ScrubGeo.Store

with SetAdditionalFields.Store

with DeleteAdditionalFields.Store

with AsyncDeleteAdditionalFields.Store

with Takedown.Store

with UpdatePossiblySensitiveTweet.Store

with AsyncUpdatePossiblySensitiveTweet.Store {

def wrap(w: TweetStore.Wrap): ManhattanTweetStore =

new TweetStoreWrapper(w, this)

with ManhattanTweetStore

with InsertTweet.StoreWrapper

with AsyncDeleteTweet.StoreWrapper

with ScrubGeo.StoreWrapper

with SetAdditionalFields.StoreWrapper

with DeleteAdditionalFields.StoreWrapper

with AsyncDeleteAdditionalFields.StoreWrapper

with Takedown.StoreWrapper

with UpdatePossiblySensitiveTweet.StoreWrapper

with AsyncUpdatePossiblySensitiveTweet.StoreWrapper

}

/\*\*

\* A TweetStore implementation that writes to Manhattan.

\*/

object ManhattanTweetStore {

val Action: AsyncWriteAction.TbirdUpdate.type = AsyncWriteAction.TbirdUpdate

private val log = Logger(getClass)

private val successResponses = Set(TweetResponseCode.Success, TweetResponseCode.Deleted)

case class AnnotationFailure(message: String) extends Exception(message)

def apply(tweetStorageClient: TweetStorageClient): ManhattanTweetStore = {

def handleStorageResponses(

responsesStitch: Stitch[Seq[TweetResponse]],

action: String

): Future[Unit] =

Stitch

.run(responsesStitch)

.onFailure {

case ex: TweetStorageException => log.warn("failed on: " + action, ex)

case \_ =>

}

.flatMap { responses =>

Future.when(responses.exists(resp => !successResponses(resp.overallResponse))) {

Future.exception(AnnotationFailure(s"$action gets failure response $responses"))

}

}

def updateTweetMediaIds(mutation: Mutation[MediaEntity]): Tweet => Tweet =

tweet => tweet.copy(media = tweet.media.map(entities => entities.map(mutation.endo)))

/\*\*

\* Does a get and set, and only sets fields that are allowed to be

\* changed. This also prevents incoming tweets containing incomplete

\* fields from being saved to Manhattan.

\*/

def updateOneTweetByIdAction(tweetId: TweetId, copyFields: Tweet => Tweet): Future[Unit] = {

Stitch.run {

tweetStorageClient.getTweet(tweetId).flatMap {

case GetTweet.Response.Found(tweet) =>

val updatedTweet = copyFields(tweet)

if (updatedTweet != tweet) {

tweetStorageClient.addTweet(updatedTweet)

} else {

Stitch.Unit

}

case \_ => Stitch.exception(UpdateTweetNotFoundException(tweetId))

}

}

}

// This should NOT be used in parallel with other write operations.

// A race condition can occur after changes to the storage library to

// return all additional fields. The resulting behavior can cause

// fields that were modified by other writes to revert to their old value.

def updateOneTweetAction(update: Tweet, copyFields: Tweet => Tweet => Tweet): Future[Unit] =

updateOneTweetByIdAction(update.id, copyFields(update))

def tweetStoreUpdateTweet(tweet: Tweet): Future[Unit] = {

val setFields = AdditionalFields.nonEmptyAdditionalFieldIds(tweet).map(Field.additionalField)

handleStorageResponses(

tweetStorageClient.updateTweet(tweet, setFields).map(Seq(\_)),

s"updateTweet($tweet, $setFields)"

)

}

// This is an edit so update the initial Tweet's control

def updateInitialTweet(event: InsertTweet.Event): Future[Unit] = {

event.initialTweetUpdateRequest match {

case Some(request) =>

updateOneTweetByIdAction(

request.initialTweetId,

tweet => InitialTweetUpdate.updateTweet(tweet, request)

)

case None => Future.Unit

}

}

new ManhattanTweetStore {

override val insertTweet: FutureEffect[InsertTweet.Event] =

FutureEffect[InsertTweet.Event] { event =>

Stitch

.run(

tweetStorageClient.addTweet(event.internalTweet.tweet)

).flatMap(\_ => updateInitialTweet(event))

}

override val asyncDeleteTweet: FutureEffect[AsyncDeleteTweet.Event] =

FutureEffect[AsyncDeleteTweet.Event] { event =>

if (event.isBounceDelete) {

Stitch.run(tweetStorageClient.bounceDelete(event.tweet.id))

} else {

Stitch.run(tweetStorageClient.softDelete(event.tweet.id))

}

}

override val retryAsyncDeleteTweet: FutureEffect[

TweetStoreRetryEvent[AsyncDeleteTweet.Event]

] =

TweetStore.retry(Action, asyncDeleteTweet)

override val scrubGeo: FutureEffect[ScrubGeo.Event] =

FutureEffect[ScrubGeo.Event] { event =>

Stitch.run(tweetStorageClient.scrub(event.tweetIds, Seq(Field.Geo)))

}

override val setAdditionalFields: FutureEffect[SetAdditionalFields.Event] =

FutureEffect[SetAdditionalFields.Event] { event =>

tweetStoreUpdateTweet(event.additionalFields)

}

override val deleteAdditionalFields: FutureEffect[DeleteAdditionalFields.Event] =

FutureEffect[DeleteAdditionalFields.Event] { event =>

handleStorageResponses(

tweetStorageClient.deleteAdditionalFields(

Seq(event.tweetId),

event.fieldIds.map(Field.additionalField)

),

s"deleteAdditionalFields(${event.tweetId}, ${event.fieldIds}})"

)

}

override val asyncDeleteAdditionalFields: FutureEffect[AsyncDeleteAdditionalFields.Event] =

FutureEffect[AsyncDeleteAdditionalFields.Event] { event =>

handleStorageResponses(

tweetStorageClient.deleteAdditionalFields(

Seq(event.tweetId),

event.fieldIds.map(Field.additionalField)

),

s"deleteAdditionalFields(Seq(${event.tweetId}), ${event.fieldIds}})"

)

}

override val retryAsyncDeleteAdditionalFields: FutureEffect[

TweetStoreRetryEvent[AsyncDeleteAdditionalFields.Event]

] =

TweetStore.retry(Action, asyncDeleteAdditionalFields)

override val takedown: FutureEffect[Takedown.Event] =

FutureEffect[Takedown.Event] { event =>

val (fieldsToUpdate, fieldsToDelete) =

Seq(

Field.TweetypieOnlyTakedownCountryCodes,

Field.TweetypieOnlyTakedownReasons

).filter(\_ => event.updateCodesAndReasons)

.partition(f => event.tweet.getFieldBlob(f.id).isDefined)

val allFieldsToUpdate = Seq(Field.HasTakedown) ++ fieldsToUpdate

Future

.join(

handleStorageResponses(

tweetStorageClient

.updateTweet(event.tweet, allFieldsToUpdate)

.map(Seq(\_)),

s"updateTweet(${event.tweet}, $allFieldsToUpdate)"

),

Future.when(fieldsToDelete.nonEmpty) {

handleStorageResponses(

tweetStorageClient

.deleteAdditionalFields(Seq(event.tweet.id), fieldsToDelete),

s"deleteAdditionalFields(Seq(${event.tweet.id}), $fieldsToDelete)"

)

}

).unit

}

override val updatePossiblySensitiveTweet: FutureEffect[UpdatePossiblySensitiveTweet.Event] =

FutureEffect[UpdatePossiblySensitiveTweet.Event] { event =>

updateOneTweetAction(event.tweet, TweetUpdate.copyNsfwFieldsForUpdate)

}

override val asyncUpdatePossiblySensitiveTweet: FutureEffect[

AsyncUpdatePossiblySensitiveTweet.Event

] =

FutureEffect[AsyncUpdatePossiblySensitiveTweet.Event] { event =>

updateOneTweetAction(event.tweet, TweetUpdate.copyNsfwFieldsForUpdate)

}

override val retryAsyncUpdatePossiblySensitiveTweet: FutureEffect[

TweetStoreRetryEvent[AsyncUpdatePossiblySensitiveTweet.Event]

] =

TweetStore.retry(Action, asyncUpdatePossiblySensitiveTweet)

}

}

}