package com.twitter.tweetypie

package handler

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.core.StoredTweetResult.\_

import com.twitter.tweetypie.core.StoredTweetResult

import com.twitter.tweetypie.core.TweetResult

import com.twitter.tweetypie.FieldId

import com.twitter.tweetypie.FutureArrow

import com.twitter.tweetypie.repository.CacheControl

import com.twitter.tweetypie.repository.TweetQuery

import com.twitter.tweetypie.repository.TweetResultRepository

import com.twitter.tweetypie.thriftscala.{BounceDeleted => BounceDeletedState}

import com.twitter.tweetypie.thriftscala.{ForceAdded => ForceAddedState}

import com.twitter.tweetypie.thriftscala.GetStoredTweetsRequest

import com.twitter.tweetypie.thriftscala.GetStoredTweetsOptions

import com.twitter.tweetypie.thriftscala.GetStoredTweetsResult

import com.twitter.tweetypie.thriftscala.{HardDeleted => HardDeletedState}

import com.twitter.tweetypie.thriftscala.{NotFound => NotFoundState}

import com.twitter.tweetypie.thriftscala.{SoftDeleted => SoftDeletedState}

import com.twitter.tweetypie.thriftscala.StatusCounts

import com.twitter.tweetypie.thriftscala.StoredTweetError

import com.twitter.tweetypie.thriftscala.StoredTweetInfo

import com.twitter.tweetypie.thriftscala.StoredTweetState

import com.twitter.tweetypie.thriftscala.{Undeleted => UndeletedState}

object GetStoredTweetsHandler {

type Type = FutureArrow[GetStoredTweetsRequest, Seq[GetStoredTweetsResult]]

def apply(tweetRepo: TweetResultRepository.Type): Type = {

FutureArrow[GetStoredTweetsRequest, Seq[GetStoredTweetsResult]] { request =>

val requestOptions: GetStoredTweetsOptions =

request.options.getOrElse(GetStoredTweetsOptions())

val queryOptions = toTweetQueryOptions(requestOptions)

val result = Stitch

.traverse(request.tweetIds) { tweetId =>

tweetRepo(tweetId, queryOptions)

.map(toStoredTweetInfo)

.map(GetStoredTweetsResult(\_))

.handle {

case \_ =>

GetStoredTweetsResult(

StoredTweetInfo(

tweetId = tweetId,

errors = Seq(StoredTweetError.FailedFetch)

)

)

}

}

Stitch.run(result)

}

}

private def toTweetQueryOptions(options: GetStoredTweetsOptions): TweetQuery.Options = {

val countsFields: Set[FieldId] = Set(

StatusCounts.FavoriteCountField.id,

StatusCounts.ReplyCountField.id,

StatusCounts.RetweetCountField.id,

StatusCounts.QuoteCountField.id

)

TweetQuery.Options(

include = GetTweetsHandler.BaseInclude.also(

tweetFields = Set(Tweet.CountsField.id) ++ options.additionalFieldIds,

countsFields = countsFields

),

cacheControl = CacheControl.NoCache,

enforceVisibilityFiltering = !options.bypassVisibilityFiltering,

forUserId = options.forUserId,

requireSourceTweet = false,

fetchStoredTweets = true

)

}

private def toStoredTweetInfo(tweetResult: TweetResult): StoredTweetInfo = {

def translateErrors(errors: Seq[StoredTweetResult.Error]): Seq[StoredTweetError] = {

errors.map {

case StoredTweetResult.Error.Corrupt => StoredTweetError.Corrupt

case StoredTweetResult.Error.FieldsMissingOrInvalid =>

StoredTweetError.FieldsMissingOrInvalid

case StoredTweetResult.Error.ScrubbedFieldsPresent => StoredTweetError.ScrubbedFieldsPresent

case StoredTweetResult.Error.ShouldBeHardDeleted => StoredTweetError.ShouldBeHardDeleted

}

}

val tweetData = tweetResult.value

tweetData.storedTweetResult match {

case Some(storedTweetResult) => {

val (tweet, storedTweetState, errors) = storedTweetResult match {

case Present(errors, \_) => (Some(tweetData.tweet), None, translateErrors(errors))

case HardDeleted(softDeletedAtMsec, hardDeletedAtMsec) =>

(

Some(tweetData.tweet),

Some(

StoredTweetState.HardDeleted(

HardDeletedState(softDeletedAtMsec, hardDeletedAtMsec))),

Seq()

)

case SoftDeleted(softDeletedAtMsec, errors, \_) =>

(

Some(tweetData.tweet),

Some(StoredTweetState.SoftDeleted(SoftDeletedState(softDeletedAtMsec))),

translateErrors(errors)

)

case BounceDeleted(deletedAtMsec, errors, \_) =>

(

Some(tweetData.tweet),

Some(StoredTweetState.BounceDeleted(BounceDeletedState(deletedAtMsec))),

translateErrors(errors)

)

case Undeleted(undeletedAtMsec, errors, \_) =>

(

Some(tweetData.tweet),

Some(StoredTweetState.Undeleted(UndeletedState(undeletedAtMsec))),

translateErrors(errors)

)

case ForceAdded(addedAtMsec, errors, \_) =>

(

Some(tweetData.tweet),

Some(StoredTweetState.ForceAdded(ForceAddedState(addedAtMsec))),

translateErrors(errors)

)

case Failed(errors) => (None, None, translateErrors(errors))

case NotFound => (None, Some(StoredTweetState.NotFound(NotFoundState())), Seq())

}

StoredTweetInfo(

tweetId = tweetData.tweet.id,

tweet = tweet.map(sanitizeNullMediaFields),

storedTweetState = storedTweetState,

errors = errors

)

}

case None =>

StoredTweetInfo(

tweetId = tweetData.tweet.id,

tweet = Some(sanitizeNullMediaFields(tweetData.tweet))

)

}

}

private def sanitizeNullMediaFields(tweet: Tweet): Tweet = {

// Some media fields are initialized as `null` at the storage layer.

// If the Tweet is meant to be hard deleted, or is not hydrated for

// some other reason but the media entities still exist, we sanitize

// these fields to allow serialization.

tweet.copy(media = tweet.media.map(\_.map { mediaEntity =>

mediaEntity.copy(

url = Option(mediaEntity.url).getOrElse(""),

mediaUrl = Option(mediaEntity.mediaUrl).getOrElse(""),

mediaUrlHttps = Option(mediaEntity.mediaUrlHttps).getOrElse(""),

displayUrl = Option(mediaEntity.displayUrl).getOrElse(""),

expandedUrl = Option(mediaEntity.expandedUrl).getOrElse(""),

)

}))

}

}