package com.twitter.tweetypie

package config

import com.twitter.servo.util.FutureArrow

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.core.\_

import com.twitter.tweetypie.handler.TweetBuilder

import com.twitter.tweetypie.handler.WritePathQueryOptions

import com.twitter.tweetypie.hydrator.EscherbirdAnnotationHydrator

import com.twitter.tweetypie.hydrator.LanguageHydrator

import com.twitter.tweetypie.hydrator.PlaceHydrator

import com.twitter.tweetypie.hydrator.ProfileGeoHydrator

import com.twitter.tweetypie.hydrator.TweetDataValueHydrator

import com.twitter.tweetypie.repository.\_

import com.twitter.tweetypie.store.InsertTweet

import com.twitter.tweetypie.store.UndeleteTweet

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.tweetypie.util.EditControlUtil

object WritePathHydration {

type HydrateQuotedTweet =

FutureArrow[(User, QuotedTweet, WritePathHydrationOptions), Option[QuoteTweetMetadata]]

case class QuoteTweetMetadata(

quotedTweet: Tweet,

quotedUser: User,

quoterHasAlreadyQuotedTweet: Boolean)

private val log = Logger(getClass)

val UserFieldsForInsert: Set[UserField] =

TweetBuilder.userFields

val AllowedMissingFieldsOnWrite: Set[FieldByPath] =

Set(

EscherbirdAnnotationHydrator.hydratedField,

LanguageHydrator.hydratedField,

PlaceHydrator.HydratedField,

ProfileGeoHydrator.hydratedField

)

/\*\*

\* Builds a FutureArrow that performs the necessary hydration in the write-path for a

\* a InsertTweet.Event. There are two separate hydration steps, pre-cache and post-cache.

\* The pre-cache hydration step performs the hydration which is safe to cache, while the

\* post-cache hydration step performs the hydration whose results we don't want to cache

\* on the tweet.

\*

\* TweetInsertEvent contains two tweet fields, `tweet` and `internalTweet`. `tweet` is

\* the input value used for hydration, and in the updated InsertTweet.Event returned by the

\* FutureArrow, `tweet` contains the post-cache hydrated tweet while `internalTweet` contains

\* the pre-cache hydrated tweet.

\*/

def hydrateInsertTweetEvent(

hydrateTweet: FutureArrow[(TweetData, TweetQuery.Options), TweetData],

hydrateQuotedTweet: HydrateQuotedTweet

): FutureArrow[InsertTweet.Event, InsertTweet.Event] =

FutureArrow { event =>

val cause = TweetQuery.Cause.Insert(event.tweet.id)

val hydrationOpts = event.hydrateOptions

val isEditControlEdit = event.tweet.editControl.exists(EditControlUtil.isEditControlEdit)

val queryOpts: TweetQuery.Options =

WritePathQueryOptions.insert(cause, event.user, hydrationOpts, isEditControlEdit)

val initTweetData =

TweetData(

tweet = event.tweet,

sourceTweetResult = event.sourceTweet.map(TweetResult(\_))

)

for {

tweetData <- hydrateTweet((initTweetData, queryOpts))

hydratedTweet = tweetData.tweet

internalTweet =

tweetData.cacheableTweetResult

.map(\_.value.toCachedTweet)

.getOrElse(

throw new IllegalStateException(s"expected cacheableTweetResult, e=${event}"))

optQt = getQuotedTweet(hydratedTweet)

.orElse(event.sourceTweet.flatMap(getQuotedTweet))

hydratedQT <- optQt match {

case None => Future.value(None)

case Some(qt) => hydrateQuotedTweet((event.user, qt, hydrationOpts))

}

} yield {

event.copy(

tweet = hydratedTweet,

\_internalTweet = Some(internalTweet),

quotedTweet = hydratedQT.map { case QuoteTweetMetadata(t, \_, \_) => t },

quotedUser = hydratedQT.map { case QuoteTweetMetadata(\_, u, \_) => u },

quoterHasAlreadyQuotedTweet = hydratedQT.exists { case QuoteTweetMetadata(\_, \_, b) => b }

)

}

}

/\*\*

\* Builds a FutureArrow for retrieving a quoted tweet metadata

\* QuotedTweet struct. If either the quoted tweet or the quoted user

\* isn't visible to the tweeting user, the FutureArrow will return None.

\*/

def hydrateQuotedTweet(

tweetRepo: TweetRepository.Optional,

userRepo: UserRepository.Optional,

quoterHasAlreadyQuotedRepo: QuoterHasAlreadyQuotedRepository.Type

): HydrateQuotedTweet = {

FutureArrow {

case (tweetingUser, qt, hydrateOptions) =>

val tweetQueryOpts = WritePathQueryOptions.quotedTweet(tweetingUser, hydrateOptions)

val userQueryOpts =

UserQueryOptions(

UserFieldsForInsert,

UserVisibility.Visible,

forUserId = Some(tweetingUser.id)

)

Stitch.run(

Stitch

.join(

tweetRepo(qt.tweetId, tweetQueryOpts),

userRepo(UserKey.byId(qt.userId), userQueryOpts),

// We're failing open here on tflock exceptions since this should not

// affect the ability to quote tweet if tflock goes down. (although if

// this call doesn't succeed, quote counts may be inaccurate for a brief

// period of time)

quoterHasAlreadyQuotedRepo(qt.tweetId, tweetingUser.id).liftToTry

)

.map {

case (Some(tweet), Some(user), isAlreadyQuoted) =>

Some(QuoteTweetMetadata(tweet, user, isAlreadyQuoted.getOrElse(false)))

case \_ => None

}

)

}

}

/\*\*

\* Builds a FutureArrow that performs any additional hydration on an UndeleteTweet.Event before

\* being passed to a TweetStore.

\*/

def hydrateUndeleteTweetEvent(

hydrateTweet: FutureArrow[(TweetData, TweetQuery.Options), TweetData],

hydrateQuotedTweet: HydrateQuotedTweet

): FutureArrow[UndeleteTweet.Event, UndeleteTweet.Event] =

FutureArrow { event =>

val cause = TweetQuery.Cause.Undelete(event.tweet.id)

val hydrationOpts = event.hydrateOptions

val isEditControlEdit = event.tweet.editControl.exists(EditControlUtil.isEditControlEdit)

val queryOpts = WritePathQueryOptions.insert(cause, event.user, hydrationOpts, isEditControlEdit)

// when undeleting a retweet, don't set sourceTweetResult to enable SourceTweetHydrator to

// hydrate it

val initTweetData = TweetData(tweet = event.tweet)

for {

tweetData <- hydrateTweet((initTweetData, queryOpts))

hydratedTweet = tweetData.tweet

internalTweet =

tweetData.cacheableTweetResult

.map(\_.value.toCachedTweet)

.getOrElse(

throw new IllegalStateException(s"expected cacheableTweetResult, e=${event}"))

optQt = getQuotedTweet(hydratedTweet)

.orElse(tweetData.sourceTweetResult.map(\_.value.tweet).flatMap(getQuotedTweet))

hydratedQt <- optQt match {

case None => Future.value(None)

case Some(qt) => hydrateQuotedTweet((event.user, qt, hydrationOpts))

}

} yield {

event.copy(

tweet = hydratedTweet,

\_internalTweet = Some(internalTweet),

sourceTweet = tweetData.sourceTweetResult.map(\_.value.tweet),

quotedTweet = hydratedQt.map { case QuoteTweetMetadata(t, \_, \_) => t },

quotedUser = hydratedQt.map { case QuoteTweetMetadata(\_, u, \_) => u },

quoterHasAlreadyQuotedTweet = hydratedQt.exists { case QuoteTweetMetadata(\_, \_, b) => b }

)

}

}

/\*\*

\* Converts a TweetDataValueHydrator into a FutureArrow that hydrates a tweet for the write-path.

\*/

def hydrateTweet(

hydrator: TweetDataValueHydrator,

stats: StatsReceiver,

allowedMissingFields: Set[FieldByPath] = AllowedMissingFieldsOnWrite

): FutureArrow[(TweetData, TweetQuery.Options), TweetData] = {

val hydrationStats = stats.scope("hydration")

val missingFieldsStats = hydrationStats.scope("missing\_fields")

FutureArrow[(TweetData, TweetQuery.Options), TweetData] {

case (td, opts) =>

Stitch

.run(hydrator(td, opts))

.rescue {

case ex =>

log.warn("Hydration failed with exception", ex)

Future.exception(

TweetHydrationError("Hydration failed with exception: " + ex, Some(ex))

)

}

.flatMap { r =>

// Record missing fields even if the request succeeds)

for (missingField <- r.state.failedFields)

missingFieldsStats.counter(missingField.fieldIdPath.mkString(".")).incr()

if ((r.state.failedFields -- allowedMissingFields).nonEmpty) {

Future.exception(

TweetHydrationError(

"Failed to hydrate. Missing Fields: " + r.state.failedFields.mkString(",")

)

)

} else {

Future.value(r.value)

}

}

}

}.trackOutcome(stats, (\_: Any) => "hydration")

}