package com.twitter.tweetypie

package store

import com.twitter.timelineservice.{thriftscala => tls}

import com.twitter.tweetypie.backends.TimelineService

import com.twitter.tweetypie.thriftscala.\_

trait TlsTimelineUpdatingStore

extends TweetStoreBase[TlsTimelineUpdatingStore]

with AsyncInsertTweet.Store

with AsyncDeleteTweet.Store

with AsyncUndeleteTweet.Store {

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

new TweetStoreWrapper(w, this)

with TlsTimelineUpdatingStore

with AsyncInsertTweet.StoreWrapper

with AsyncDeleteTweet.StoreWrapper

with AsyncUndeleteTweet.StoreWrapper

}

/\*\*

\* An implementation of TweetStore that sends update events to

\* the Timeline Service.

\*/

object TlsTimelineUpdatingStore {

val Action: AsyncWriteAction.TimelineUpdate.type = AsyncWriteAction.TimelineUpdate

/\*\*

\* Converts a TweetyPie Tweet to tls.Tweet

\*

\* @param explicitCreatedAt when Some, overrides the default getTimestamp defined in package

\* object com.twitter.tweetypie

\*/

def tweetToTLSFullTweet(

hasMedia: Tweet => Boolean

)(

tweet: Tweet,

explicitCreatedAt: Option[Time],

noteTweetMentionedUserIds: Option[Seq[Long]]

): tls.FullTweet =

tls.FullTweet(

userId = getUserId(tweet),

tweetId = tweet.id,

mentionedUserIds =

noteTweetMentionedUserIds.getOrElse(getMentions(tweet).flatMap(\_.userId)).toSet,

isNullcasted = TweetLenses.nullcast.get(tweet),

conversationId = TweetLenses.conversationId.get(tweet).getOrElse(tweet.id),

narrowcastGeos = Set.empty,

createdAtMs = explicitCreatedAt.getOrElse(getTimestamp(tweet)).inMillis,

hasMedia = hasMedia(tweet),

directedAtUserId = TweetLenses.directedAtUser.get(tweet).map(\_.userId),

retweet = getShare(tweet).map { share =>

tls.Retweet(

sourceUserId = share.sourceUserId,

sourceTweetId = share.sourceStatusId,

parentTweetId = Some(share.parentStatusId)

)

},

reply = getReply(tweet).map { reply =>

tls.Reply(

inReplyToUserId = reply.inReplyToUserId,

inReplyToTweetId = reply.inReplyToStatusId

)

},

quote = tweet.quotedTweet.map { qt =>

tls.Quote(

quotedUserId = qt.userId,

quotedTweetId = qt.tweetId

)

},

mediaTags = tweet.mediaTags,

text = Some(getText(tweet))

)

val logger: Logger = Logger(getClass)

def logValidationFailed(stats: StatsReceiver): tls.ProcessEventResult => Unit = {

case tls.ProcessEventResult(tls.ProcessEventResultType.ValidationFailed, errors) =>

logger.error(s"Validation Failed in processEvent2: $errors")

stats.counter("processEvent2\_validation\_failed").incr()

case \_ => ()

}

def apply(

processEvent2: TimelineService.ProcessEvent2,

hasMedia: Tweet => Boolean,

stats: StatsReceiver

): TlsTimelineUpdatingStore = {

val toTlsTweet = tweetToTLSFullTweet(hasMedia) \_

val processAndLog =

processEvent2.andThen(FutureArrow.fromFunction(logValidationFailed(stats)))

new TlsTimelineUpdatingStore {

override val asyncInsertTweet: FutureEffect[AsyncInsertTweet.Event] =

processAndLog

.contramap[AsyncInsertTweet.Event] { event =>

tls.Event.FullTweetCreate(

tls.FullTweetCreateEvent(

toTlsTweet(event.tweet, Some(event.timestamp), event.noteTweetMentionedUserIds),

event.timestamp.inMillis,

featureContext = event.featureContext

)

)

}

.asFutureEffect[AsyncInsertTweet.Event]

override val retryAsyncInsertTweet: FutureEffect[

TweetStoreRetryEvent[AsyncInsertTweet.Event]

] =

TweetStore.retry(Action, asyncInsertTweet)

override val asyncUndeleteTweet: FutureEffect[AsyncUndeleteTweet.Event] =

processAndLog

.contramap[AsyncUndeleteTweet.Event] { event =>

tls.Event.FullTweetRestore(

tls.FullTweetRestoreEvent(

toTlsTweet(event.tweet, None, None),

event.deletedAt.map(\_.inMillis)

)

)

}

.asFutureEffect[AsyncUndeleteTweet.Event]

override val retryAsyncUndeleteTweet: FutureEffect[

TweetStoreRetryEvent[AsyncUndeleteTweet.Event]

] =

TweetStore.retry(Action, asyncUndeleteTweet)

override val asyncDeleteTweet: FutureEffect[AsyncDeleteTweet.Event] =

processAndLog

.contramap[AsyncDeleteTweet.Event] { event =>

tls.Event.FullTweetDelete(

tls.FullTweetDeleteEvent(

toTlsTweet(event.tweet, None, None),

event.timestamp.inMillis,

isUserErasure = Some(event.isUserErasure),

isBounceDelete = Some(event.isBounceDelete)

)

)

}

.asFutureEffect[AsyncDeleteTweet.Event]

override val retryAsyncDeleteTweet: FutureEffect[

TweetStoreRetryEvent[AsyncDeleteTweet.Event]

] =

TweetStore.retry(Action, asyncDeleteTweet)

}

}

}