package com.twitter.tweetypie

package store

import com.twitter.context.thriftscala.FeatureContext

import com.twitter.tweetypie.core.GeoSearchRequestId

import com.twitter.tweetypie.store.TweetEventDataScrubber.scrub

import com.twitter.tweetypie.thriftscala.\_

object InsertTweet extends TweetStore.SyncModule {

case class Event(

tweet: Tweet,

user: User,

timestamp: Time,

\_internalTweet: Option[CachedTweet] = None,

sourceTweet: Option[Tweet] = None,

sourceUser: Option[User] = None,

quotedTweet: Option[Tweet] = None,

quotedUser: Option[User] = None,

parentUserId: Option[UserId] = None,

initialTweetUpdateRequest: Option[InitialTweetUpdateRequest] = None,

dark: Boolean = false,

hydrateOptions: WritePathHydrationOptions = WritePathHydrationOptions(),

featureContext: Option[FeatureContext] = None,

geoSearchRequestId: Option[GeoSearchRequestId] = None,

additionalContext: Option[collection.Map[TweetCreateContextKey, String]] = None,

transientContext: Option[TransientCreateContext] = None,

quoterHasAlreadyQuotedTweet: Boolean = false,

noteTweetMentionedUserIds: Option[Seq[Long]] = None)

extends SyncTweetStoreEvent("insert\_tweet")

with QuotedTweetOps {

def internalTweet: CachedTweet =

\_internalTweet.getOrElse(

throw new IllegalStateException(

s"internalTweet should have been set in WritePathHydration, ${this}"

)

)

def toAsyncRequest(

scrubUser: User => User,

scrubSourceTweet: Tweet => Tweet,

scrubSourceUser: User => User

): AsyncInsertRequest =

AsyncInsertRequest(

tweet = tweet,

cachedTweet = internalTweet,

user = scrubUser(user),

sourceTweet = sourceTweet.map(scrubSourceTweet),

sourceUser = sourceUser.map(scrubSourceUser),

quotedTweet = quotedTweet.map(scrubSourceTweet),

quotedUser = quotedUser.map(scrubSourceUser),

parentUserId = parentUserId,

featureContext = featureContext,

timestamp = timestamp.inMillis,

geoSearchRequestId = geoSearchRequestId.map(\_.requestID),

additionalContext = additionalContext,

transientContext = transientContext,

quoterHasAlreadyQuotedTweet = Some(quoterHasAlreadyQuotedTweet),

initialTweetUpdateRequest = initialTweetUpdateRequest,

noteTweetMentionedUserIds = noteTweetMentionedUserIds

)

}

trait Store {

val insertTweet: FutureEffect[Event]

}

trait StoreWrapper extends Store { self: TweetStoreWrapper[Store] =>

override val insertTweet: FutureEffect[Event] = wrap(underlying.insertTweet)

}

object Store {

def apply(

logLensStore: LogLensStore,

manhattanStore: ManhattanTweetStore,

tweetStatsStore: TweetStatsStore,

cachingTweetStore: CachingTweetStore,

limiterStore: LimiterStore,

asyncEnqueueStore: AsyncEnqueueStore,

userCountsUpdatingStore: GizmoduckUserCountsUpdatingStore,

tweetCountsUpdatingStore: TweetCountsCacheUpdatingStore

): Store =

new Store {

override val insertTweet: FutureEffect[Event] =

FutureEffect.sequentially(

logLensStore.insertTweet,

manhattanStore.insertTweet,

tweetStatsStore.insertTweet,

FutureEffect.inParallel(

// allow write-through caching to fail without failing entire insert

cachingTweetStore.ignoreFailures.insertTweet,

limiterStore.ignoreFailures.insertTweet,

asyncEnqueueStore.insertTweet,

userCountsUpdatingStore.insertTweet,

tweetCountsUpdatingStore.insertTweet

)

)

}

}

}

object AsyncInsertTweet extends TweetStore.AsyncModule {

private val log = Logger(getClass)

object Event {

def fromAsyncRequest(request: AsyncInsertRequest): TweetStoreEventOrRetry[Event] =

TweetStoreEventOrRetry(

Event(

tweet = request.tweet,

cachedTweet = request.cachedTweet,

user = request.user,

optUser = Some(request.user),

timestamp = Time.fromMilliseconds(request.timestamp),

sourceTweet = request.sourceTweet,

sourceUser = request.sourceUser,

parentUserId = request.parentUserId,

featureContext = request.featureContext,

quotedTweet = request.quotedTweet,

quotedUser = request.quotedUser,

geoSearchRequestId = request.geoSearchRequestId,

additionalContext = request.additionalContext,

transientContext = request.transientContext,

quoterHasAlreadyQuotedTweet = request.quoterHasAlreadyQuotedTweet.getOrElse(false),

initialTweetUpdateRequest = request.initialTweetUpdateRequest,

noteTweetMentionedUserIds = request.noteTweetMentionedUserIds

),

request.retryAction,

RetryEvent

)

}

case class Event(

tweet: Tweet,

cachedTweet: CachedTweet,

user: User,

optUser: Option[User],

timestamp: Time,

sourceTweet: Option[Tweet] = None,

sourceUser: Option[User] = None,

parentUserId: Option[UserId] = None,

featureContext: Option[FeatureContext] = None,

quotedTweet: Option[Tweet] = None,

quotedUser: Option[User] = None,

geoSearchRequestId: Option[String] = None,

additionalContext: Option[collection.Map[TweetCreateContextKey, String]] = None,

transientContext: Option[TransientCreateContext] = None,

quoterHasAlreadyQuotedTweet: Boolean = false,

initialTweetUpdateRequest: Option[InitialTweetUpdateRequest] = None,

noteTweetMentionedUserIds: Option[Seq[Long]] = None)

extends AsyncTweetStoreEvent("async\_insert\_tweet")

with QuotedTweetOps

with TweetStoreTweetEvent {

def toAsyncRequest(action: Option[AsyncWriteAction] = None): AsyncInsertRequest =

AsyncInsertRequest(

tweet = tweet,

cachedTweet = cachedTweet,

user = user,

sourceTweet = sourceTweet,

sourceUser = sourceUser,

parentUserId = parentUserId,

retryAction = action,

featureContext = featureContext,

timestamp = timestamp.inMillis,

quotedTweet = quotedTweet,

quotedUser = quotedUser,

geoSearchRequestId = geoSearchRequestId,

additionalContext = additionalContext,

transientContext = transientContext,

quoterHasAlreadyQuotedTweet = Some(quoterHasAlreadyQuotedTweet),

initialTweetUpdateRequest = initialTweetUpdateRequest,

noteTweetMentionedUserIds = noteTweetMentionedUserIds

)

override def toTweetEventData: Seq[TweetEventData] =

Seq(

TweetEventData.TweetCreateEvent(

TweetCreateEvent(

tweet = scrub(tweet),

user = user,

sourceUser = sourceUser,

sourceTweet = sourceTweet.map(scrub),

retweetParentUserId = parentUserId,

quotedTweet = publicQuotedTweet.map(scrub),

quotedUser = publicQuotedUser,

additionalContext = additionalContext,

transientContext = transientContext,

quoterHasAlreadyQuotedTweet = Some(quoterHasAlreadyQuotedTweet)

)

)

)

override def enqueueRetry(service: ThriftTweetService, action: AsyncWriteAction): Future[Unit] =

service.asyncInsert(toAsyncRequest(Some(action)))

}

case class RetryEvent(action: AsyncWriteAction, event: Event)

extends TweetStoreRetryEvent[Event] {

override val eventType: AsyncWriteEventType.Insert.type = AsyncWriteEventType.Insert

override val scribedTweetOnFailure: Option[Tweet] = Some(event.tweet)

}

trait Store {

val asyncInsertTweet: FutureEffect[Event]

val retryAsyncInsertTweet: FutureEffect[TweetStoreRetryEvent[Event]]

}

trait StoreWrapper extends Store { self: TweetStoreWrapper[Store] =>

override val asyncInsertTweet: FutureEffect[Event] = wrap(underlying.asyncInsertTweet)

override val retryAsyncInsertTweet: FutureEffect[TweetStoreRetryEvent[Event]] = wrap(

underlying.retryAsyncInsertTweet)

}

object Store {

def apply(

replicatingStore: ReplicatingTweetStore,

indexingStore: TweetIndexingStore,

tweetCountsUpdatingStore: TweetCountsCacheUpdatingStore,

timelineUpdatingStore: TlsTimelineUpdatingStore,

eventBusEnqueueStore: TweetEventBusStore,

fanoutServiceStore: FanoutServiceStore,

scribeMediaTagStore: ScribeMediaTagStore,

userGeotagUpdateStore: GizmoduckUserGeotagUpdateStore,

geoSearchRequestIDStore: GeoSearchRequestIDStore

): Store = {

val stores: Seq[Store] =

Seq(

replicatingStore,

indexingStore,

timelineUpdatingStore,

eventBusEnqueueStore,

fanoutServiceStore,

userGeotagUpdateStore,

tweetCountsUpdatingStore,

scribeMediaTagStore,

geoSearchRequestIDStore

)

def build[E <: TweetStoreEvent](extract: Store => FutureEffect[E]): FutureEffect[E] =

FutureEffect.inParallel[E](stores.map(extract): \_\*)

new Store {

override val asyncInsertTweet: FutureEffect[Event] = build(\_.asyncInsertTweet)

override val retryAsyncInsertTweet: FutureEffect[TweetStoreRetryEvent[Event]] = build(

\_.retryAsyncInsertTweet)

}

}

}

}

object ReplicatedInsertTweet extends TweetStore.ReplicatedModule {

case class Event(

tweet: Tweet,

cachedTweet: CachedTweet,

quoterHasAlreadyQuotedTweet: Boolean = false,

initialTweetUpdateRequest: Option[InitialTweetUpdateRequest] = None)

extends ReplicatedTweetStoreEvent("replicated\_insert\_tweet")

trait Store {

val replicatedInsertTweet: FutureEffect[Event]

}

trait StoreWrapper extends Store { self: TweetStoreWrapper[Store] =>

override val replicatedInsertTweet: FutureEffect[Event] = wrap(underlying.replicatedInsertTweet)

}

object Store {

def apply(

cachingTweetStore: CachingTweetStore,

tweetCountsUpdatingStore: TweetCountsCacheUpdatingStore

): Store = {

new Store {

override val replicatedInsertTweet: FutureEffect[Event] =

FutureEffect.inParallel(

cachingTweetStore.replicatedInsertTweet,

tweetCountsUpdatingStore.replicatedInsertTweet.ignoreFailures

)

}

}

}

}