package com.twitter.tweetypie

package store

import com.twitter.tweetypie.thriftscala.\_

object UpdatePossiblySensitiveTweet extends TweetStore.SyncModule {

case class Event(

tweet: Tweet,

user: User,

timestamp: Time,

byUserId: UserId,

nsfwAdminChange: Option[Boolean],

nsfwUserChange: Option[Boolean],

note: Option[String],

host: Option[String])

extends SyncTweetStoreEvent("update\_possibly\_sensitive\_tweet") {

def toAsyncRequest: AsyncUpdatePossiblySensitiveTweetRequest =

AsyncUpdatePossiblySensitiveTweetRequest(

tweet = tweet,

user = user,

byUserId = byUserId,

timestamp = timestamp.inMillis,

nsfwAdminChange = nsfwAdminChange,

nsfwUserChange = nsfwUserChange,

note = note,

host = host

)

}

trait Store {

val updatePossiblySensitiveTweet: FutureEffect[Event]

}

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

override val updatePossiblySensitiveTweet: FutureEffect[Event] = wrap(

underlying.updatePossiblySensitiveTweet

)

}

object Store {

def apply(

manhattanStore: ManhattanTweetStore,

cachingTweetStore: CachingTweetStore,

logLensStore: LogLensStore,

asyncEnqueueStore: AsyncEnqueueStore

): Store =

new Store {

override val updatePossiblySensitiveTweet: FutureEffect[Event] =

FutureEffect.inParallel(

manhattanStore.ignoreFailures.updatePossiblySensitiveTweet,

cachingTweetStore.ignoreFailures.updatePossiblySensitiveTweet,

logLensStore.updatePossiblySensitiveTweet,

asyncEnqueueStore.updatePossiblySensitiveTweet

)

}

}

}

object AsyncUpdatePossiblySensitiveTweet extends TweetStore.AsyncModule {

object Event {

def fromAsyncRequest(

request: AsyncUpdatePossiblySensitiveTweetRequest

): TweetStoreEventOrRetry[Event] =

TweetStoreEventOrRetry(

AsyncUpdatePossiblySensitiveTweet.Event(

tweet = request.tweet,

user = request.user,

optUser = Some(request.user),

timestamp = Time.fromMilliseconds(request.timestamp),

byUserId = request.byUserId,

nsfwAdminChange = request.nsfwAdminChange,

nsfwUserChange = request.nsfwUserChange,

note = request.note,

host = request.host

),

request.action,

RetryEvent

)

}

case class Event(

tweet: Tweet,

user: User,

optUser: Option[User],

timestamp: Time,

byUserId: UserId,

nsfwAdminChange: Option[Boolean],

nsfwUserChange: Option[Boolean],

note: Option[String],

host: Option[String])

extends AsyncTweetStoreEvent("async\_update\_possibly\_sensitive\_tweet")

with TweetStoreTweetEvent {

def toAsyncRequest(

action: Option[AsyncWriteAction] = None

): AsyncUpdatePossiblySensitiveTweetRequest =

AsyncUpdatePossiblySensitiveTweetRequest(

tweet = tweet,

user = user,

byUserId = byUserId,

timestamp = timestamp.inMillis,

nsfwAdminChange = nsfwAdminChange,

nsfwUserChange = nsfwUserChange,

note = note,

host = host,

action = action

)

override def toTweetEventData: Seq[TweetEventData] =

Seq(

TweetEventData.TweetPossiblySensitiveUpdateEvent(

TweetPossiblySensitiveUpdateEvent(

tweetId = tweet.id,

userId = user.id,

nsfwAdmin = TweetLenses.nsfwAdmin.get(tweet),

nsfwUser = TweetLenses.nsfwUser.get(tweet)

)

)

)

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

service.asyncUpdatePossiblySensitiveTweet(toAsyncRequest(Some(action)))

}

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

extends TweetStoreRetryEvent[Event] {

override val eventType: AsyncWriteEventType.UpdatePossiblySensitiveTweet.type =

AsyncWriteEventType.UpdatePossiblySensitiveTweet

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

}

trait Store {

val asyncUpdatePossiblySensitiveTweet: FutureEffect[Event]

val retryAsyncUpdatePossiblySensitiveTweet: FutureEffect[TweetStoreRetryEvent[Event]]

}

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

override val asyncUpdatePossiblySensitiveTweet: FutureEffect[Event] = wrap(

underlying.asyncUpdatePossiblySensitiveTweet

)

override val retryAsyncUpdatePossiblySensitiveTweet: FutureEffect[TweetStoreRetryEvent[Event]] =

wrap(

underlying.retryAsyncUpdatePossiblySensitiveTweet

)

}

object Store {

def apply(

manhattanStore: ManhattanTweetStore,

cachingTweetStore: CachingTweetStore,

replicatingStore: ReplicatingTweetStore,

guanoStore: GuanoServiceStore,

eventBusStore: TweetEventBusStore

): Store = {

val stores: Seq[Store] =

Seq(

manhattanStore,

cachingTweetStore,

replicatingStore,

guanoStore,

eventBusStore

)

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

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

new Store {

override val asyncUpdatePossiblySensitiveTweet: FutureEffect[Event] = build(

\_.asyncUpdatePossiblySensitiveTweet)

override val retryAsyncUpdatePossiblySensitiveTweet: FutureEffect[

TweetStoreRetryEvent[Event]

] = build(

\_.retryAsyncUpdatePossiblySensitiveTweet

)

}

}

}

}

object ReplicatedUpdatePossiblySensitiveTweet extends TweetStore.ReplicatedModule {

case class Event(tweet: Tweet)

extends ReplicatedTweetStoreEvent("replicated\_update\_possibly\_sensitive\_tweet")

trait Store {

val replicatedUpdatePossiblySensitiveTweet: FutureEffect[Event]

}

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

override val replicatedUpdatePossiblySensitiveTweet: FutureEffect[Event] = wrap(

underlying.replicatedUpdatePossiblySensitiveTweet

)

}

object Store {

def apply(cachingTweetStore: CachingTweetStore): Store = {

new Store {

override val replicatedUpdatePossiblySensitiveTweet: FutureEffect[Event] =

cachingTweetStore.replicatedUpdatePossiblySensitiveTweet

}

}

}

}