package com.twitter.tweetypie

package store

import com.twitter.takedown.util.TakedownReasons

import com.twitter.tseng.withholding.thriftscala.TakedownReason

import com.twitter.tweetypie.thriftscala.\_

object Takedown extends TweetStore.SyncModule {

case class Event(

tweet: Tweet, // for CachingTweetStore / ManhattanTweetStore / ReplicatedTakedown

timestamp: Time,

user: Option[User] = None,

takedownReasons: Seq[TakedownReason] = Seq(), // for EventBus

reasonsToAdd: Seq[TakedownReason] = Seq(), // for Guano

reasonsToRemove: Seq[TakedownReason] = Seq(), // for Guano

auditNote: Option[String] = None,

host: Option[String] = None,

byUserId: Option[UserId] = None,

eventbusEnqueue: Boolean = true,

scribeForAudit: Boolean = true,

// If ManhattanTweetStore should update countryCodes and reasons

updateCodesAndReasons: Boolean = false)

extends SyncTweetStoreEvent("takedown") {

def toAsyncRequest(): AsyncTakedownRequest =

AsyncTakedownRequest(

tweet = tweet,

user = user,

takedownReasons = takedownReasons,

reasonsToAdd = reasonsToAdd,

reasonsToRemove = reasonsToRemove,

scribeForAudit = scribeForAudit,

eventbusEnqueue = eventbusEnqueue,

auditNote = auditNote,

byUserId = byUserId,

host = host,

timestamp = timestamp.inMillis

)

}

trait Store {

val takedown: FutureEffect[Event]

}

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

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

}

object Store {

def apply(

logLensStore: LogLensStore,

manhattanStore: ManhattanTweetStore,

cachingTweetStore: CachingTweetStore,

asyncEnqueueStore: AsyncEnqueueStore

): Store =

new Store {

override val takedown: FutureEffect[Event] =

FutureEffect.inParallel(

logLensStore.takedown,

FutureEffect.sequentially(

manhattanStore.takedown,

FutureEffect.inParallel(

cachingTweetStore.takedown,

asyncEnqueueStore.takedown

)

)

)

}

}

}

object AsyncTakedown extends TweetStore.AsyncModule {

object Event {

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

TweetStoreEventOrRetry(

Event(

tweet = request.tweet,

optUser = request.user,

takedownReasons = request.takedownReasons,

reasonsToAdd = request.reasonsToAdd,

reasonsToRemove = request.reasonsToRemove,

auditNote = request.auditNote,

host = request.host,

byUserId = request.byUserId,

eventbusEnqueue = request.eventbusEnqueue,

scribeForAudit = request.scribeForAudit,

timestamp = Time.fromMilliseconds(request.timestamp)

),

request.retryAction,

RetryEvent

)

}

case class Event(

tweet: Tweet,

timestamp: Time,

optUser: Option[User],

takedownReasons: Seq[TakedownReason], // for EventBus

reasonsToAdd: Seq[TakedownReason], // for Guano

reasonsToRemove: Seq[TakedownReason], // for Guano

auditNote: Option[String], // for Guano

host: Option[String], // for Guano

byUserId: Option[UserId], // for Guano

eventbusEnqueue: Boolean,

scribeForAudit: Boolean)

extends AsyncTweetStoreEvent("async\_takedown")

with TweetStoreTweetEvent {

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

AsyncTakedownRequest(

tweet = tweet,

user = optUser,

takedownReasons = takedownReasons,

reasonsToAdd = reasonsToAdd,

reasonsToRemove = reasonsToRemove,

scribeForAudit = scribeForAudit,

eventbusEnqueue = eventbusEnqueue,

auditNote = auditNote,

byUserId = byUserId,

host = host,

timestamp = timestamp.inMillis,

retryAction = action

)

override def toTweetEventData: Seq[TweetEventData] =

optUser.map { user =>

TweetEventData.TweetTakedownEvent(

TweetTakedownEvent(

tweetId = tweet.id,

userId = user.id,

takedownCountryCodes =

takedownReasons.collect(TakedownReasons.reasonToCountryCode).sorted,

takedownReasons = takedownReasons

)

)

}.toSeq

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

service.asyncTakedown(toAsyncRequest(Some(action)))

}

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

extends TweetStoreRetryEvent[Event] {

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

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

}

trait Store {

val asyncTakedown: FutureEffect[Event]

val retryAsyncTakedown: FutureEffect[TweetStoreRetryEvent[Event]]

}

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

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

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

underlying.retryAsyncTakedown)

}

object Store {

def apply(

replicatingStore: ReplicatingTweetStore,

guanoStore: GuanoServiceStore,

eventBusEnqueueStore: TweetEventBusStore

): Store = {

val stores: Seq[Store] =

Seq(

replicatingStore,

guanoStore,

eventBusEnqueueStore

)

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

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

new Store {

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

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

\_.retryAsyncTakedown)

}

}

}

}

object ReplicatedTakedown extends TweetStore.ReplicatedModule {

case class Event(tweet: Tweet) extends ReplicatedTweetStoreEvent("takedown")

trait Store {

val replicatedTakedown: FutureEffect[Event]

}

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

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

}

object Store {

def apply(cachingTweetStore: CachingTweetStore): Store = {

new Store {

override val replicatedTakedown: FutureEffect[Event] = cachingTweetStore.replicatedTakedown

}

}

}

}