package com.twitter.tweetypie

package store

import com.twitter.tweetypie.thriftscala.\_

object SetRetweetVisibility extends TweetStore.SyncModule {

case class Event(

retweetId: TweetId,

visible: Boolean,

srcId: TweetId,

retweetUserId: UserId,

srcTweetUserId: UserId,

timestamp: Time)

extends SyncTweetStoreEvent("set\_retweet\_visibility") {

def toAsyncRequest: AsyncSetRetweetVisibilityRequest =

AsyncSetRetweetVisibilityRequest(

retweetId = retweetId,

visible = visible,

srcId = srcId,

retweetUserId = retweetUserId,

sourceTweetUserId = srcTweetUserId,

timestamp = timestamp.inMillis

)

}

trait Store {

val setRetweetVisibility: FutureEffect[Event]

}

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

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

}

object Store {

/\*\*

\* [[AsyncEnqueueStore]] - use this store to call the asyncSetRetweetVisibility endpoint.

\*

\* @see [[AsyncSetRetweetVisibility.Store.apply]]

\*/

def apply(asyncEnqueueStore: AsyncEnqueueStore): Store =

new Store {

override val setRetweetVisibility: FutureEffect[Event] =

asyncEnqueueStore.setRetweetVisibility

}

}

}

object AsyncSetRetweetVisibility extends TweetStore.AsyncModule {

case class Event(

retweetId: TweetId,

visible: Boolean,

srcId: TweetId,

retweetUserId: UserId,

srcTweetUserId: UserId,

timestamp: Time)

extends AsyncTweetStoreEvent("async\_set\_retweet\_visibility") {

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

AsyncSetRetweetVisibilityRequest(

retweetId = retweetId,

visible = visible,

srcId = srcId,

retweetUserId = retweetUserId,

sourceTweetUserId = srcTweetUserId,

retryAction = action,

timestamp = timestamp.inMillis

)

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

service.asyncSetRetweetVisibility(toAsyncRequest(Some(action)))

}

object Event {

def fromAsyncRequest(req: AsyncSetRetweetVisibilityRequest): TweetStoreEventOrRetry[Event] =

TweetStoreEventOrRetry(

AsyncSetRetweetVisibility.Event(

retweetId = req.retweetId,

visible = req.visible,

srcId = req.srcId,

retweetUserId = req.retweetUserId,

srcTweetUserId = req.sourceTweetUserId,

timestamp = Time.fromMilliseconds(req.timestamp)

),

req.retryAction,

RetryEvent

)

}

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

extends TweetStoreRetryEvent[Event] {

override val eventType: AsyncWriteEventType.SetRetweetVisibility.type =

AsyncWriteEventType.SetRetweetVisibility

override val scribedTweetOnFailure: None.type = None

}

trait Store {

val asyncSetRetweetVisibility: FutureEffect[Event]

val retryAsyncSetRetweetVisibility: FutureEffect[TweetStoreRetryEvent[Event]]

}

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

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

val retryAsyncSetRetweetVisibility: FutureEffect[TweetStoreRetryEvent[Event]] = wrap(

underlying.retryAsyncSetRetweetVisibility)

}

object Store {

/\*\*

\* [[TweetIndexingStore]] - archive or unarchive a retweet edge in TFlock RetweetGraph

\* [[TweetCountsCacheUpdatingStore]] - modify the retweet count directly in cache.

\* [[ReplicatingTweetStore]] - replicate this [[Event]] in the other DC.

\* [[RetweetArchivalEnqueueStore]] - publish RetweetArchivalEvent to "retweet\_archival\_events" event stream.

\*

\* @see [[ReplicatedSetRetweetVisibility.Store.apply]]

\*/

def apply(

tweetIndexingStore: TweetIndexingStore,

tweetCountsCacheUpdatingStore: TweetCountsCacheUpdatingStore,

replicatingTweetStore: ReplicatingTweetStore,

retweetArchivalEnqueueStore: RetweetArchivalEnqueueStore

): Store = {

val stores: Seq[Store] =

Seq(

tweetIndexingStore,

tweetCountsCacheUpdatingStore,

replicatingTweetStore,

retweetArchivalEnqueueStore

)

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

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

new Store {

override val asyncSetRetweetVisibility: FutureEffect[Event] = build(

\_.asyncSetRetweetVisibility)

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

build(\_.retryAsyncSetRetweetVisibility)

}

}

}

}

object ReplicatedSetRetweetVisibility extends TweetStore.ReplicatedModule {

case class Event(srcId: TweetId, visible: Boolean)

extends ReplicatedTweetStoreEvent("replicated\_set\_retweet\_visibility")

trait Store {

val replicatedSetRetweetVisibility: FutureEffect[Event]

}

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

override val replicatedSetRetweetVisibility: FutureEffect[Event] =

wrap(underlying.replicatedSetRetweetVisibility)

}

object Store {

/\*\*

\* [[TweetCountsCacheUpdatingStore]] - replicate modifying the retweet count directly in cache.

\*/

def apply(tweetCountsCacheUpdatingStore: TweetCountsCacheUpdatingStore): Store =

new Store {

override val replicatedSetRetweetVisibility: FutureEffect[Event] =

tweetCountsCacheUpdatingStore.replicatedSetRetweetVisibility

}

}

}