package com.twitter.tweetypie

package store

import com.twitter.finagle.service.RetryPolicy

import com.twitter.finagle.stats.Stat

import com.twitter.servo.util.RetryHandler

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.util.Timer

object TweetStore {

// Using the old-school c.t.logging.Logger here as this log is only used by

// servo.FutureEffect's trackOutcome method, which needs that kind of logger.

val log: com.twitter.logging.Logger = com.twitter.logging.Logger(getClass)

/\*\*

\* Adapts a tweet store on a specific TweetStoreEvent type to one that handles

\* TweetStoreRetryEvents of that type that match the given AsyncWriteAction.

\*/

def retry[T <: AsyncTweetStoreEvent](

action: AsyncWriteAction,

store: FutureEffect[T]

): FutureEffect[TweetStoreRetryEvent[T]] =

store.contramap[TweetStoreRetryEvent[T]](\_.event).onlyIf(\_.action == action)

/\*\*

\* Defines an abstract polymorphic operation to be applied to FutureEffects over any

\* TweetStoreEvent type. The Wrap operation is defined over all possible

\* FutureEffect[E <: TweetStoreEvent] types.

\*/

trait Wrap {

def apply[E <: TweetStoreEvent](handler: FutureEffect[E]): FutureEffect[E]

}

/\*\*

\* A Wrap operation that applies standardized metrics collection to the FutureEffect.

\*/

case class Tracked(stats: StatsReceiver) extends Wrap {

def apply[E <: TweetStoreEvent](handler: FutureEffect[E]): FutureEffect[E] =

FutureEffect[E] { event =>

Stat.timeFuture(stats.scope(event.name).stat("latency\_ms")) {

handler(event)

}

}.trackOutcome(stats, \_.name, log)

}

/\*\*

\* A Wrap operation that makes the FutureEffect enabled according to the given gate.

\*/

case class Gated(gate: Gate[Unit]) extends Wrap {

def apply[E <: TweetStoreEvent](handler: FutureEffect[E]): FutureEffect[E] =

handler.enabledBy(gate)

}

/\*\*

\* A Wrap operation that updates the FutureEffect to ignore failures.

\*/

object IgnoreFailures extends Wrap {

def apply[E <: TweetStoreEvent](handler: FutureEffect[E]): FutureEffect[E] =

handler.ignoreFailures

}

/\*\*

\* A Wrap operation that updates the FutureEffect to ignore failures upon completion.

\*/

object IgnoreFailuresUponCompletion extends Wrap {

def apply[E <: TweetStoreEvent](handler: FutureEffect[E]): FutureEffect[E] =

handler.ignoreFailuresUponCompletion

}

/\*\*

\* A Wrap operation that applies a RetryHandler to FutureEffects.

\*/

case class Retry(retryHandler: RetryHandler[Unit]) extends Wrap {

def apply[E <: TweetStoreEvent](handler: FutureEffect[E]): FutureEffect[E] =

handler.retry(retryHandler)

}

/\*\*

\* A Wrap operation that applies a RetryHandler to FutureEffects.

\*/

case class ReplicatedEventRetry(retryHandler: RetryHandler[Unit]) extends Wrap {

def apply[E <: TweetStoreEvent](handler: FutureEffect[E]): FutureEffect[E] =

FutureEffect[E] { event =>

event.retryStrategy match {

case TweetStoreEvent.ReplicatedEventLocalRetry => handler.retry(retryHandler)(event)

case \_ => handler(event)

}

}

}

/\*\*

\* A Wrap operation that configures async-retry behavior to async-write events.

\*/

class AsyncRetry(

localRetryPolicy: RetryPolicy[Try[Nothing]],

enqueueRetryPolicy: RetryPolicy[Try[Nothing]],

timer: Timer,

tweetService: ThriftTweetService,

scribe: FutureEffect[FailedAsyncWrite]

)(

stats: StatsReceiver,

action: AsyncWriteAction)

extends Wrap {

override def apply[E <: TweetStoreEvent](handler: FutureEffect[E]): FutureEffect[E] =

FutureEffect[E] { event =>

event.retryStrategy match {

case TweetStoreEvent.EnqueueAsyncRetry(enqueueRetry) =>

enqueueAsyncRetry(handler, enqueueRetry)(event)

case TweetStoreEvent.LocalRetryThenScribeFailure(toFailedAsyncWrite) =>

localRetryThenScribeFailure(handler, toFailedAsyncWrite)(event)

case \_ =>

handler(event)

}

}

private def enqueueAsyncRetry[E <: TweetStoreEvent](

handler: FutureEffect[E],

enqueueRetry: (ThriftTweetService, AsyncWriteAction) => Future[Unit]

): FutureEffect[E] = {

val retryInitCounter = stats.counter("retries\_initiated")

// enqueues failed TweetStoreEvents to the deferredrpc-backed tweetService

// to be retried. this store uses the enqueueRetryPolicy to retry the enqueue

// attempts in the case of deferredrpc application failures.

val enqueueRetryHandler =

FutureEffect[E](\_ => enqueueRetry(tweetService, action))

.retry(RetryHandler.failuresOnly(enqueueRetryPolicy, timer, stats.scope("enqueue\_retry")))

handler.rescue {

case ex =>

TweetStore.log.warning(ex, s"will retry $action")

retryInitCounter.incr()

enqueueRetryHandler

}

}

private def localRetryThenScribeFailure[E <: TweetStoreEvent](

handler: FutureEffect[E],

toFailedAsyncWrite: AsyncWriteAction => FailedAsyncWrite

): FutureEffect[E] = {

val exhaustedCounter = stats.counter("retries\_exhausted")

// scribe events that failed after exhausting all retries

val scribeEventHandler =

FutureEffect[E](\_ => scribe(toFailedAsyncWrite(action)))

// wraps `handle` with a retry policy to retry failures with a backoff. if we exhaust

// all retries, then we pass the event to `scribeEventStore` to scribe the failure.

handler

.retry(RetryHandler.failuresOnly(localRetryPolicy, timer, stats))

.rescue {

case ex =>

TweetStore.log.warning(ex, s"exhausted retries on $action")

exhaustedCounter.incr()

scribeEventHandler

}

}

}

/\*\*

\* Parent trait for defining a "module" that defines a TweetStoreEvent type and corresponding

\* TweetStore and TweetStoreWrapper types.

\*/

sealed trait Module {

type Store

type StoreWrapper <: Store

}

/\*\*

\* Parent trait for defining a "module" that defines a sync TweetStoreEvent.

\*/

trait SyncModule extends Module {

type Event <: SyncTweetStoreEvent

}

/\*\*

\* Parent trait for defining a "module" that defines an async TweetStoreEvent and a

\* TweetStoreRetryEvent.

\*/

trait AsyncModule extends Module {

type Event <: AsyncTweetStoreEvent

type RetryEvent <: TweetStoreRetryEvent[Event]

}

/\*\*

\* Parent trait for defining a "module" that defines a replicated TweetStoreEvent.

\*/

trait ReplicatedModule extends Module {

type Event <: ReplicatedTweetStoreEvent

}

}

/\*\*

\* Trait for TweetStore implementations that support handler wrapping.

\*/

trait TweetStoreBase[Self] {

import TweetStore.\_

/\*\*

\* Returns a new store of type Self with Wrap applied to each event handler in this instance.

\*/

def wrap(w: Wrap): Self

/\*\*

\* Applies the Tracked Wrap operation to the store.

\*/

def tracked(stats: StatsReceiver): Self = wrap(Tracked(stats))

/\*\*

\* Applies the Gated Wrap operation to the store.

\*/

def enabledBy(gate: Gate[Unit]): Self = wrap(Gated(gate))

/\*\*

\* Applies the IgnoreFailures Wrap operation to the store.

\*/

def ignoreFailures: Self = wrap(IgnoreFailures)

/\*\*

\* Applies the IgnoreFailuresUponCompletion Wrap operation to the store.

\*/

def ignoreFailuresUponCompletion: Self = wrap(IgnoreFailuresUponCompletion)

/\*\*

\* Applies a RetryHandler to each event handler.

\*/

def retry(retryHandler: RetryHandler[Unit]): Self = wrap(Retry(retryHandler))

/\*\*

\* Applies a RetryHandler to replicated event handlers.

\*/

def replicatedRetry(retryHandler: RetryHandler[Unit]): Self =

wrap(ReplicatedEventRetry(retryHandler))

/\*\*

\* Applies the AsyncRetryConfig Wrap operation to the store.

\*/

def asyncRetry(cfg: AsyncRetry): Self = wrap(cfg)

}

/\*\*

\* An abstract base class for tweet store instances that wrap another tweet store instance.

\* You can mix event-specific store wrapper traits into this class to automatically

\* have the event-specific handlers wrapped.

\*/

abstract class TweetStoreWrapper[+T](

protected val wrap: TweetStore.Wrap,

protected val underlying: T)

/\*\*

\* A TweetStore that has a handler for all possible TweetStoreEvents.

\*/

trait TotalTweetStore

extends AsyncDeleteAdditionalFields.Store

with AsyncDeleteTweet.Store

with AsyncIncrBookmarkCount.Store

with AsyncIncrFavCount.Store

with AsyncInsertTweet.Store

with AsyncSetAdditionalFields.Store

with AsyncSetRetweetVisibility.Store

with AsyncTakedown.Store

with AsyncUndeleteTweet.Store

with AsyncUpdatePossiblySensitiveTweet.Store

with DeleteAdditionalFields.Store

with DeleteTweet.Store

with Flush.Store

with IncrBookmarkCount.Store

with IncrFavCount.Store

with InsertTweet.Store

with QuotedTweetDelete.Store

with QuotedTweetTakedown.Store

with ReplicatedDeleteAdditionalFields.Store

with ReplicatedDeleteTweet.Store

with ReplicatedIncrBookmarkCount.Store

with ReplicatedIncrFavCount.Store

with ReplicatedInsertTweet.Store

with ReplicatedScrubGeo.Store

with ReplicatedSetAdditionalFields.Store

with ReplicatedSetRetweetVisibility.Store

with ReplicatedTakedown.Store

with ReplicatedUndeleteTweet.Store

with ReplicatedUpdatePossiblySensitiveTweet.Store

with ScrubGeo.Store

with ScrubGeoUpdateUserTimestamp.Store

with SetAdditionalFields.Store

with SetRetweetVisibility.Store

with Takedown.Store

with UndeleteTweet.Store

with UpdatePossiblySensitiveTweet.Store