package com.twitter.tweetypie

package store

import com.twitter.finagle.tracing.Trace

import com.twitter.tweetypie.store.TweetStoreEvent.RetryStrategy

import com.twitter.tweetypie.thriftscala.\_

object TweetStoreEvent {

/\*\*

\* Parent trait for indicating what type of retry strategy to apply to event handlers

\* for the corresponding event type. Different classes of events use different strategies.

\*/

sealed trait RetryStrategy

/\*\*

\* Indicates that the event type doesn't support retries.

\*/

case object NoRetry extends RetryStrategy

/\*\*

\* Indicates that if an event handler encounters a failure, it should enqueue a

\* retry to be performed asynchronously.

\*/

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

extends RetryStrategy

/\*\*

\* Indicates that if an event handler encounters a failure, it should retry

\* the event locally some number of times, before eventually given up and scribing

\* the failure.

\*/

case class LocalRetryThenScribeFailure(toFailedAsyncWrite: AsyncWriteAction => FailedAsyncWrite)

extends RetryStrategy

/\*\*

\* Indicates that if an event handler encounters a failure, it should retry

\* the event locally some number of times.

\*/

case object ReplicatedEventLocalRetry extends RetryStrategy

}

/\*\*

\* The abstract parent class for all TweetStoreEvent types.

\*/

sealed trait TweetStoreEvent {

val name: String

val traceId: Long = Trace.id.traceId.toLong

/\*\*

\* Indicates a particular retry behavior that should be applied to event handlers for

\* the corresponding event type. The specifics of the strategy might depend upon the

\* specific TweetStore implementation.

\*/

def retryStrategy: RetryStrategy

}

abstract class SyncTweetStoreEvent(val name: String) extends TweetStoreEvent {

override def retryStrategy: RetryStrategy = TweetStoreEvent.NoRetry

}

abstract class AsyncTweetStoreEvent(val name: String) extends TweetStoreEvent {

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

override def retryStrategy: RetryStrategy = TweetStoreEvent.EnqueueAsyncRetry(enqueueRetry)

}

abstract class ReplicatedTweetStoreEvent(val name: String) extends TweetStoreEvent {

override def retryStrategy: RetryStrategy = TweetStoreEvent.ReplicatedEventLocalRetry

}

/\*\*

\* A trait for all TweetStoreEvents that become TweetEvents.

\*/

trait TweetStoreTweetEvent {

val timestamp: Time

val optUser: Option[User]

/\*\*

\* Most TweetStoreTweetEvents map to a single TweetEvent, but some

\* optionally map to an event and others map to multiple events, so

\* this method needs to return a Seq of TweetEventData.

\*/

def toTweetEventData: Seq[TweetEventData]

}

/\*\*

\* The abstract parent class for an event that indicates a particular action

\* for a particular event that needs to be retried via the async-write-retrying mechanism.

\*/

abstract class TweetStoreRetryEvent[E <: AsyncTweetStoreEvent] extends TweetStoreEvent {

override val name = "async\_write\_retry"

def action: AsyncWriteAction

def event: E

def eventType: AsyncWriteEventType

def scribedTweetOnFailure: Option[Tweet]

override def retryStrategy: RetryStrategy =

TweetStoreEvent.LocalRetryThenScribeFailure(action =>

FailedAsyncWrite(eventType, action, scribedTweetOnFailure))

}

/\*\*

\* Functions as a disjunction between an event type E and it's corresonding

\* retry event type TweetStoreRetryEvent[E]

\*/

case class TweetStoreEventOrRetry[E <: AsyncTweetStoreEvent](

event: E,

toRetry: Option[TweetStoreRetryEvent[E]]) {

def toInitial: Option[E] = if (retryAction.isDefined) None else Some(event)

def retryAction: Option[RetryStrategy] = toRetry.map(\_.retryStrategy)

def hydrate(f: E => Future[E]): Future[TweetStoreEventOrRetry[E]] =

f(event).map(e => copy(event = e))

}

object TweetStoreEventOrRetry {

def apply[E <: AsyncTweetStoreEvent, R <: TweetStoreRetryEvent[E]](

event: E,

retryAction: Option[AsyncWriteAction],

toRetryEvent: (AsyncWriteAction, E) => R

): TweetStoreEventOrRetry[E] =

TweetStoreEventOrRetry(event, retryAction.map(action => toRetryEvent(action, event)))

object First {

/\*\* matches against TweetStoreEventOrRetry instances for an initial event \*/

def unapply[E <: AsyncTweetStoreEvent](it: TweetStoreEventOrRetry[E]): Option[E] =

it.toInitial

}

object Retry {

/\*\* matches against TweetStoreEventOrRetry instances for a retry event \*/

def unapply[E <: AsyncTweetStoreEvent](

it: TweetStoreEventOrRetry[E]

): Option[TweetStoreRetryEvent[E]] =

it.toRetry

}

}