package com.twitter.tweetypie

package backends

import com.twitter.finagle.Backoff

import com.twitter.finagle.service.RetryPolicy

import com.twitter.flockdb.client.{thriftscala => flockdb, \_}

import com.twitter.servo

import com.twitter.servo.util.RetryHandler

import com.twitter.tweetypie.core.OverCapacity

import com.twitter.tweetypie.util.RetryPolicyBuilder

import com.twitter.util.Future

import com.twitter.util.TimeoutException

object TFlock {

val log = Logger(this.getClass)

case class Config(

requestTimeout: Duration,

timeoutBackoffs: Stream[Duration],

flockExceptionBackoffs: Stream[Duration],

overCapacityBackoffs: Stream[Duration],

defaultPageSize: Int = 1000) {

def apply(svc: flockdb.FlockDB.MethodPerEndpoint, ctx: Backend.Context): TFlockClient = {

val retryHandler =

RetryHandler[Any](

retryPolicy(timeoutBackoffs, flockExceptionBackoffs, overCapacityBackoffs),

ctx.timer,

ctx.stats

)

val rescueHandler = translateExceptions.andThen(Future.exception)

val exceptionCounter = new servo.util.ExceptionCounter(ctx.stats, "failures")

val timeoutException = new TimeoutException(s"tflock: $requestTimeout")

val wrapper =

new WrappingFunction {

def apply[T](f: => Future[T]): Future[T] =

retryHandler {

exceptionCounter {

f.raiseWithin(ctx.timer, requestTimeout, timeoutException)

.onFailure(logFlockExceptions)

.rescue(rescueHandler)

}

}

}

val wrappedClient = new WrappingFlockClient(svc, wrapper, wrapper)

val statsClient = new StatsCollectingFlockService(wrappedClient, ctx.stats)

new TFlockClient(statsClient, defaultPageSize)

}

}

def isOverCapacity(ex: flockdb.FlockException): Boolean =

ex.errorCode match {

case Some(flockdb.Constants.READ\_OVERCAPACITY\_ERROR) => true

case Some(flockdb.Constants.WRITE\_OVERCAPACITY\_ERROR) => true

case \_ => false

}

/\*\*

\* Builds a RetryPolicy for tflock operations that will retry timeouts with the specified

\* timeout backoffs, and will retry non-overcapacity FlockExceptions with the

\* specified flockExceptionBackoffs backoffs, and will retry over-capacity exceptions with

\* the specified overCapacityBackoffs.

\*/

def retryPolicy(

timeoutBackoffs: Stream[Duration],

flockExceptionBackoffs: Stream[Duration],

overCapacityBackoffs: Stream[Duration]

): RetryPolicy[Try[Any]] =

RetryPolicy.combine[Try[Any]](

RetryPolicyBuilder.timeouts[Any](timeoutBackoffs),

RetryPolicy.backoff(Backoff.fromStream(flockExceptionBackoffs)) {

case Throw(ex: flockdb.FlockException) if !isOverCapacity(ex) => true

case Throw(\_: flockdb.FlockQuotaException) => false

},

RetryPolicy.backoff(Backoff.fromStream(overCapacityBackoffs)) {

case Throw(ex: flockdb.FlockException) if isOverCapacity(ex) => true

case Throw(\_: flockdb.FlockQuotaException) => true

case Throw(\_: OverCapacity) => true

}

)

val logFlockExceptions: Throwable => Unit = {

case t: flockdb.FlockException => {

log.info("FlockException from TFlock", t)

}

case \_ =>

}

/\*\*

\* Converts FlockExceptions with overcapacity codes into tweetypie's OverCapacity.

\*/

val translateExceptions: PartialFunction[Throwable, Throwable] = {

case t: flockdb.FlockQuotaException =>

OverCapacity(s"tflock: throttled ${t.description}")

case t: flockdb.FlockException if isOverCapacity(t) =>

OverCapacity(s"tflock: ${t.description}")

}

}