package com.twitter.tweetypie

package backends

import com.twitter.finagle.Backoff

import com.twitter.finagle.service.RetryPolicy

import com.twitter.servo.util.FutureArrow

import com.twitter.timelineservice.thriftscala.Event

import com.twitter.timelineservice.thriftscala.PerspectiveQuery

import com.twitter.timelineservice.thriftscala.PerspectiveResult

import com.twitter.timelineservice.thriftscala.ProcessEventResult

import com.twitter.timelineservice.thriftscala.StatusTimelineResult

import com.twitter.timelineservice.thriftscala.TimelineQuery

import com.twitter.timelineservice.{thriftscala => tls}

import com.twitter.tweetypie.util.RetryPolicyBuilder

object TimelineService {

import Backend.\_

type GetStatusTimeline = FutureArrow[Seq[tls.TimelineQuery], Seq[tls.StatusTimelineResult]]

type GetPerspectives = FutureArrow[Seq[tls.PerspectiveQuery], Seq[tls.PerspectiveResult]]

type ProcessEvent2 = FutureArrow[tls.Event, tls.ProcessEventResult]

private val warmupQuery =

// we need a non-empty query, since tls treats empty queries as an error

tls.TimelineQuery(

timelineType = tls.TimelineType.User,

timelineId = 620530287L, // same user id that timelineservice-api uses for warmup

maxCount = 1

)

def fromClient(client: tls.TimelineService.MethodPerEndpoint): TimelineService =

new TimelineService {

val processEvent2 = FutureArrow(client.processEvent2 \_)

val getStatusTimeline = FutureArrow(client.getStatusTimeline \_)

val getPerspectives = FutureArrow(client.getPerspectives \_)

def ping(): Future[Unit] =

client.touchTimeline(Seq(warmupQuery)).handle { case \_: tls.InternalServerError => }

}

case class Config(writeRequestPolicy: Policy, readRequestPolicy: Policy) {

def apply(svc: TimelineService, ctx: Backend.Context): TimelineService = {

val build = new PolicyAdvocate("TimelineService", ctx, svc)

new TimelineService {

val processEvent2: FutureArrow[Event, ProcessEventResult] =

build("processEvent2", writeRequestPolicy, \_.processEvent2)

val getStatusTimeline: FutureArrow[Seq[TimelineQuery], Seq[StatusTimelineResult]] =

build("getStatusTimeline", readRequestPolicy, \_.getStatusTimeline)

val getPerspectives: FutureArrow[Seq[PerspectiveQuery], Seq[PerspectiveResult]] =

build("getPerspectives", readRequestPolicy, \_.getPerspectives)

def ping(): Future[Unit] = svc.ping()

}

}

}

case class FailureBackoffsPolicy(

timeoutBackoffs: Stream[Duration] = Stream.empty,

tlsExceptionBackoffs: Stream[Duration] = Stream.empty)

extends Policy {

def toFailureRetryPolicy: FailureRetryPolicy =

FailureRetryPolicy(

RetryPolicy.combine(

RetryPolicyBuilder.timeouts(timeoutBackoffs),

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

case Throw(ex: tls.InternalServerError) => true

}

)

)

def apply[A, B](name: String, ctx: Context): Builder[A, B] =

toFailureRetryPolicy(name, ctx)

}

implicit val warmup: Warmup[TimelineService] =

Warmup[TimelineService]("timelineservice")(\_.ping())

}

trait TimelineService {

import TimelineService.\_

val processEvent2: ProcessEvent2

val getStatusTimeline: GetStatusTimeline

val getPerspectives: GetPerspectives

def ping(): Future[Unit]

}