package com.twitter.timelineranker.client

import com.twitter.finagle.SourcedException

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.timelineranker.{thriftscala => thrift}

import com.twitter.timelineranker.model.\_

import com.twitter.timelines.util.stats.RequestStats

import com.twitter.timelines.util.stats.RequestStatsReceiver

import com.twitter.util.Future

import com.twitter.util.Return

import com.twitter.util.Throw

import com.twitter.util.Try

case class TimelineRankerException(message: String)

extends Exception(message)

with SourcedException {

serviceName = "timelineranker"

}

/\*\*

\* A timeline ranker client whose methods accept and produce model object instances

\* instead of thrift instances.

\*/

class TimelineRankerClient(

private val client: thrift.TimelineRanker.MethodPerEndpoint,

statsReceiver: StatsReceiver)

extends RequestStats {

private[this] val baseScope = statsReceiver.scope("timelineRankerClient")

private[this] val timelinesRequestStats = RequestStatsReceiver(baseScope.scope("timelines"))

private[this] val recycledTweetRequestStats = RequestStatsReceiver(

baseScope.scope("recycledTweet"))

private[this] val recapHydrationRequestStats = RequestStatsReceiver(

baseScope.scope("recapHydration"))

private[this] val recapAuthorRequestStats = RequestStatsReceiver(baseScope.scope("recapAuthor"))

private[this] val entityTweetsRequestStats = RequestStatsReceiver(baseScope.scope("entityTweets"))

private[this] val utegLikedByTweetsRequestStats = RequestStatsReceiver(

baseScope.scope("utegLikedByTweets"))

private[this] def fetchRecapQueryResultHead(

results: Seq[Try[CandidateTweetsResult]]

): CandidateTweetsResult = {

results.head match {

case Return(result) => result

case Throw(e) => throw e

}

}

private[this] def tryResults[Req, Rep](

reqs: Seq[Req],

stats: RequestStatsReceiver,

findError: Req => Option[thrift.TimelineError],

)(

getRep: (Req, RequestStatsReceiver) => Try[Rep]

): Seq[Try[Rep]] = {

reqs.map { req =>

findError(req) match {

case Some(error) if error.reason.exists { \_ == thrift.ErrorReason.OverCapacity } =>

// bubble up over capacity error, server shall handle it

stats.onFailure(error)

Throw(error)

case Some(error) =>

stats.onFailure(error)

Throw(TimelineRankerException(error.message))

case None =>

getRep(req, stats)

}

}

}

private[this] def tryCandidateTweetsResults(

responses: Seq[thrift.GetCandidateTweetsResponse],

requestScopedStats: RequestStatsReceiver

): Seq[Try[CandidateTweetsResult]] = {

def errorInResponse(

response: thrift.GetCandidateTweetsResponse

): Option[thrift.TimelineError] = {

response.error

}

tryResults(

responses,

requestScopedStats,

errorInResponse

) { (response, stats) =>

stats.onSuccess()

Return(CandidateTweetsResult.fromThrift(response))

}

}

def getTimeline(query: TimelineQuery): Future[Try[Timeline]] = {

getTimelines(Seq(query)).map(\_.head)

}

def getTimelines(queries: Seq[TimelineQuery]): Future[Seq[Try[Timeline]]] = {

def errorInResponse(response: thrift.GetTimelineResponse): Option[thrift.TimelineError] = {

response.error

}

val thriftQueries = queries.map(\_.toThrift)

timelinesRequestStats.latency {

client.getTimelines(thriftQueries).map { responses =>

tryResults(

responses,

timelinesRequestStats,

errorInResponse

) { (response, stats) =>

response.timeline match {

case Some(timeline) =>

stats.onSuccess()

Return(Timeline.fromThrift(timeline))

// Should not really happen.

case None =>

val tlrException =

TimelineRankerException("No timeline returned even when no error occurred.")

stats.onFailure(tlrException)

Throw(tlrException)

}

}

}

}

}

def getRecycledTweetCandidates(query: RecapQuery): Future[CandidateTweetsResult] = {

getRecycledTweetCandidates(Seq(query)).map(fetchRecapQueryResultHead)

}

def getRecycledTweetCandidates(

queries: Seq[RecapQuery]

): Future[Seq[Try[CandidateTweetsResult]]] = {

val thriftQueries = queries.map(\_.toThriftRecapQuery)

recycledTweetRequestStats.latency {

client.getRecycledTweetCandidates(thriftQueries).map {

tryCandidateTweetsResults(\_, recycledTweetRequestStats)

}

}

}

def hydrateTweetCandidates(query: RecapQuery): Future[CandidateTweetsResult] = {

hydrateTweetCandidates(Seq(query)).map(fetchRecapQueryResultHead)

}

def hydrateTweetCandidates(queries: Seq[RecapQuery]): Future[Seq[Try[CandidateTweetsResult]]] = {

val thriftQueries = queries.map(\_.toThriftRecapHydrationQuery)

recapHydrationRequestStats.latency {

client.hydrateTweetCandidates(thriftQueries).map {

tryCandidateTweetsResults(\_, recapHydrationRequestStats)

}

}

}

def getRecapCandidatesFromAuthors(query: RecapQuery): Future[CandidateTweetsResult] = {

getRecapCandidatesFromAuthors(Seq(query)).map(fetchRecapQueryResultHead)

}

def getRecapCandidatesFromAuthors(

queries: Seq[RecapQuery]

): Future[Seq[Try[CandidateTweetsResult]]] = {

val thriftQueries = queries.map(\_.toThriftRecapQuery)

recapAuthorRequestStats.latency {

client.getRecapCandidatesFromAuthors(thriftQueries).map {

tryCandidateTweetsResults(\_, recapAuthorRequestStats)

}

}

}

def getEntityTweetCandidates(query: RecapQuery): Future[CandidateTweetsResult] = {

getEntityTweetCandidates(Seq(query)).map(fetchRecapQueryResultHead)

}

def getEntityTweetCandidates(

queries: Seq[RecapQuery]

): Future[Seq[Try[CandidateTweetsResult]]] = {

val thriftQueries = queries.map(\_.toThriftEntityTweetsQuery)

entityTweetsRequestStats.latency {

client.getEntityTweetCandidates(thriftQueries).map {

tryCandidateTweetsResults(\_, entityTweetsRequestStats)

}

}

}

def getUtegLikedByTweetCandidates(query: RecapQuery): Future[CandidateTweetsResult] = {

getUtegLikedByTweetCandidates(Seq(query)).map(fetchRecapQueryResultHead)

}

def getUtegLikedByTweetCandidates(

queries: Seq[RecapQuery]

): Future[Seq[Try[CandidateTweetsResult]]] = {

val thriftQueries = queries.map(\_.toThriftUtegLikedByTweetsQuery)

utegLikedByTweetsRequestStats.latency {

client.getUtegLikedByTweetCandidates(thriftQueries).map {

tryCandidateTweetsResults(\_, utegLikedByTweetsRequestStats)

}

}

}

}