/\*\* Copyright 2012 Twitter, Inc. \*/

package com.twitter.tweetypie.service

import com.twitter.coreservices.StratoPublicApiRequestAttributionCounter

import com.twitter.finagle.CancelledRequestException

import com.twitter.finagle.context.Contexts

import com.twitter.finagle.context.Deadline

import com.twitter.finagle.mux.ClientDiscardedRequestException

import com.twitter.finagle.stats.DefaultStatsReceiver

import com.twitter.finagle.stats.Stat

import com.twitter.servo.exception.thriftscala.ClientError

import com.twitter.servo.util.ExceptionCategorizer

import com.twitter.servo.util.MemoizedExceptionCounterFactory

import com.twitter.tweetypie.Future

import com.twitter.tweetypie.Gate

import com.twitter.tweetypie.Logger

import com.twitter.tweetypie.StatsReceiver

import com.twitter.tweetypie.ThriftTweetService

import com.twitter.tweetypie.TweetId

import com.twitter.tweetypie.client\_id.ClientIdHelper

import com.twitter.tweetypie.context.TweetypieContext

import com.twitter.tweetypie.core.OverCapacity

import com.twitter.tweetypie.serverutil.ExceptionCounter

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.util.Promise

/\*\*

\* A TweetService that takes care of the handling of requests from

\* external services. In particular, this wrapper doesn't have any

\* logic for handling requests itself. It just serves as a gateway for

\* requests and responses, making sure that the underlying tweet

\* service only sees requests it should handle and that the external

\* clients get clean responses.

\*

\* - Ensures that exceptions are propagated cleanly

\* - Sheds traffic if necessary

\* - Authenticates clients

\* - Records stats about clients

\*

\* For each endpoint, we record both client-specific and total metrics for number of requests,

\* successes, exceptions, and latency. The stats names follow the patterns:

\* - ./<methodName>/requests

\* - ./<methodName>/success

\* - ./<methodName>/client\_errors

\* - ./<methodName>/server\_errors

\* - ./<methodName>/exceptions

\* - ./<methodName>/exceptions/<exceptionName>

\* - ./<methodName>/<clientId>/requests

\* - ./<methodName>/<clientId>/success

\* - ./<methodName>/<clientId>/exceptions

\* - ./<methodName>/<clientId>/exceptions/<exceptionName>

\*/

class ClientHandlingTweetService(

underlying: ThriftTweetService,

stats: StatsReceiver,

loadShedEligible: Gate[String],

shedReadTrafficVoluntarily: Gate[Unit],

requestAuthorizer: ClientRequestAuthorizer,

getTweetsAuthorizer: MethodAuthorizer[GetTweetsRequest],

getTweetFieldsAuthorizer: MethodAuthorizer[GetTweetFieldsRequest],

requestSizeAuthorizer: MethodAuthorizer[Int],

clientIdHelper: ClientIdHelper)

extends ThriftTweetService {

import RescueExceptions.\_

private val log = Logger("com.twitter.tweetypie.service.TweetService")

private[this] val Requests = "requests"

private[this] val Success = "success"

private[this] val Latency = "latency\_ms"

private[this] val StratoStatsCounter = new StratoPublicApiRequestAttributionCounter(

DefaultStatsReceiver

)

private[this] val clientServerCategorizer =

ExceptionCategorizer.simple {

\_ match {

case \_: ClientError | \_: AccessDenied => "client\_errors"

case \_ => "server\_errors"

}

}

private[this] val preServoExceptionCountersWithClientId =

new MemoizedExceptionCounterFactory(stats)

private[this] val preServoExceptionCounters =

new MemoizedExceptionCounterFactory(stats, categorizer = ExceptionCounter.defaultCategorizer)

private[this] val postServoExceptionCounters =

new MemoizedExceptionCounterFactory(stats, categorizer = clientServerCategorizer)

private def clientId: String =

clientIdHelper.effectiveClientId.getOrElse(ClientIdHelper.UnknownClientId)

private def clientIdRoot: String =

clientIdHelper.effectiveClientIdRoot.getOrElse(ClientIdHelper.UnknownClientId)

private[this] val futureOverCapacityException =

Future.exception(OverCapacity("Request rejected due to load shedding."))

private[this] def ifNotOverCapacityRead[T](

methodStats: StatsReceiver,

requestSize: Long

)(

f: => Future[T]

): Future[T] = {

val couldShed = loadShedEligible(clientId)

val doShed = couldShed && shedReadTrafficVoluntarily()

methodStats.stat("loadshed\_incoming\_requests").add(requestSize)

if (couldShed) {

methodStats.stat("loadshed\_eligible\_requests").add(requestSize)

} else {

methodStats.stat("loadshed\_ineligible\_requests").add(requestSize)

}

if (doShed) {

methodStats.stat("loadshed\_rejected\_requests").add(requestSize)

futureOverCapacityException

} else {

f

}

}

private def maybeTimeFuture[A](maybeStat: Option[Stat])(f: => Future[A]) =

maybeStat match {

case Some(stat) => Stat.timeFuture(stat)(f)

case None => f

}

/\*\*

\* Perform the action, increment the appropriate counters, and clean up the exceptions to servo exceptions

\*

\* This method also masks all interrupts to prevent request cancellation on hangup.

\*/

private[this] def trackS[T](

name: String,

requestInfo: Any,

extraStatPrefix: Option[String] = None,

requestSize: Option[Long] = None

)(

action: StatsReceiver => Future[T]

): Future[T] = {

val methodStats = stats.scope(name)

val clientStats = methodStats.scope(clientIdRoot)

val cancelledCounter = methodStats.counter("cancelled")

/\*\*

\* Returns an identical future except that it ignores interrupts and increments a counter

\* when a request is cancelled. This is [[Future]].masked but with a counter.

\*/

def maskedWithStats[A](f: Future[A]): Future[A] = {

val p = Promise[A]()

p.setInterruptHandler {

case \_: ClientDiscardedRequestException | \_: CancelledRequestException =>

cancelledCounter.incr()

}

f.proxyTo(p)

p

}

maskedWithStats(

requestAuthorizer(name, clientIdHelper.effectiveClientId)

.flatMap { \_ =>

methodStats.counter(Requests).incr()

extraStatPrefix.foreach(p => methodStats.counter(p, Requests).incr())

clientStats.counter(Requests).incr()

StratoStatsCounter.recordStats(name, "tweets", requestSize.getOrElse(1L))

Stat.timeFuture(methodStats.stat(Latency)) {

Stat.timeFuture(clientStats.stat(Latency)) {

maybeTimeFuture(extraStatPrefix.map(p => methodStats.stat(p, Latency))) {

TweetypieContext.Local.trackStats(stats, methodStats, clientStats)

// Remove the deadline for backend requests when we mask client cancellations so

// that side-effects are applied to all backend services even after client timeouts.

// Wrap and then flatten an extra layer of Future to capture any thrown exceptions.

Future(Contexts.broadcast.letClear(Deadline)(action(methodStats))).flatten

}

}

}

}

).onSuccess { \_ =>

methodStats.counter(Success).incr()

extraStatPrefix.foreach(p => methodStats.counter(p, Success).incr())

clientStats.counter(Success).incr()

}

.onFailure { e =>

preServoExceptionCounters(name)(e)

preServoExceptionCountersWithClientId(name, clientIdRoot)(e)

}

.rescue(rescueToServoFailure(name, clientId))

.onFailure { e =>

postServoExceptionCounters(name)(e)

logFailure(e, requestInfo)

}

}

def track[T](

name: String,

requestInfo: Any,

extraStatPrefix: Option[String] = None,

requestSize: Option[Long] = None

)(

action: => Future[T]

): Future[T] = {

trackS(name, requestInfo, extraStatPrefix, requestSize) { \_: StatsReceiver => action }

}

private def logFailure(ex: Throwable, requestInfo: Any): Unit =

log.warn(s"Returning failure response: $ex\n failed request info: $requestInfo")

object RequestWidthPrefix {

private def prefix(width: Int) = {

val bucketMin =

width match {

case c if c < 10 => "0\_9"

case c if c < 100 => "10\_99"

case \_ => "100\_plus"

}

s"width\_$bucketMin"

}

def forGetTweetsRequest(r: GetTweetsRequest): String = prefix(r.tweetIds.size)

def forGetTweetFieldsRequest(r: GetTweetFieldsRequest): String = prefix(r.tweetIds.size)

}

object WithMediaPrefix {

def forPostTweetRequest(r: PostTweetRequest): String =

if (r.mediaUploadIds.exists(\_.nonEmpty))

"with\_media"

else

"without\_media"

}

override def getTweets(request: GetTweetsRequest): Future[Seq[GetTweetResult]] =

trackS(

"get\_tweets",

request,

Some(RequestWidthPrefix.forGetTweetsRequest(request)),

Some(request.tweetIds.size)

) { stats =>

getTweetsAuthorizer(request, clientId).flatMap { \_ =>

ifNotOverCapacityRead(stats, request.tweetIds.length) {

underlying.getTweets(request)

}

}

}

override def getTweetFields(request: GetTweetFieldsRequest): Future[Seq[GetTweetFieldsResult]] =

trackS(

"get\_tweet\_fields",

request,

Some(RequestWidthPrefix.forGetTweetFieldsRequest(request)),

Some(request.tweetIds.size)

) { stats =>

getTweetFieldsAuthorizer(request, clientId).flatMap { \_ =>

ifNotOverCapacityRead(stats, request.tweetIds.length) {

underlying.getTweetFields(request)

}

}

}

override def replicatedGetTweets(request: GetTweetsRequest): Future[Unit] =

track("replicated\_get\_tweets", request, requestSize = Some(request.tweetIds.size)) {

underlying.replicatedGetTweets(request).rescue {

case e: Throwable => Future.Unit // do not need deferredrpc to retry on exceptions

}

}

override def replicatedGetTweetFields(request: GetTweetFieldsRequest): Future[Unit] =

track("replicated\_get\_tweet\_fields", request, requestSize = Some(request.tweetIds.size)) {

underlying.replicatedGetTweetFields(request).rescue {

case e: Throwable => Future.Unit // do not need deferredrpc to retry on exceptions

}

}

override def getTweetCounts(request: GetTweetCountsRequest): Future[Seq[GetTweetCountsResult]] =

trackS("get\_tweet\_counts", request, requestSize = Some(request.tweetIds.size)) { stats =>

ifNotOverCapacityRead(stats, request.tweetIds.length) {

requestSizeAuthorizer(request.tweetIds.size, clientId).flatMap { \_ =>

underlying.getTweetCounts(request)

}

}

}

override def replicatedGetTweetCounts(request: GetTweetCountsRequest): Future[Unit] =

track("replicated\_get\_tweet\_counts", request, requestSize = Some(request.tweetIds.size)) {

underlying.replicatedGetTweetCounts(request).rescue {

case e: Throwable => Future.Unit // do not need deferredrpc to retry on exceptions

}

}

override def postTweet(request: PostTweetRequest): Future[PostTweetResult] =

track("post\_tweet", request, Some(WithMediaPrefix.forPostTweetRequest(request))) {

underlying.postTweet(request)

}

override def postRetweet(request: RetweetRequest): Future[PostTweetResult] =

track("post\_retweet", request) {

underlying.postRetweet(request)

}

override def setAdditionalFields(request: SetAdditionalFieldsRequest): Future[Unit] =

track("set\_additional\_fields", request) {

underlying.setAdditionalFields(request)

}

override def deleteAdditionalFields(request: DeleteAdditionalFieldsRequest): Future[Unit] =

track("delete\_additional\_fields", request, requestSize = Some(request.tweetIds.size)) {

requestSizeAuthorizer(request.tweetIds.size, clientId).flatMap { \_ =>

underlying.deleteAdditionalFields(request)

}

}

override def asyncSetAdditionalFields(request: AsyncSetAdditionalFieldsRequest): Future[Unit] =

track("async\_set\_additional\_fields", request) {

underlying.asyncSetAdditionalFields(request)

}

override def asyncDeleteAdditionalFields(

request: AsyncDeleteAdditionalFieldsRequest

): Future[Unit] =

track("async\_delete\_additional\_fields", request) {

underlying.asyncDeleteAdditionalFields(request)

}

override def replicatedUndeleteTweet2(request: ReplicatedUndeleteTweet2Request): Future[Unit] =

track("replicated\_undelete\_tweet2", request) { underlying.replicatedUndeleteTweet2(request) }

override def replicatedInsertTweet2(request: ReplicatedInsertTweet2Request): Future[Unit] =

track("replicated\_insert\_tweet2", request) { underlying.replicatedInsertTweet2(request) }

override def asyncInsert(request: AsyncInsertRequest): Future[Unit] =

track("async\_insert", request) { underlying.asyncInsert(request) }

override def updatePossiblySensitiveTweet(

request: UpdatePossiblySensitiveTweetRequest

): Future[Unit] =

track("update\_possibly\_sensitive\_tweet", request) {

underlying.updatePossiblySensitiveTweet(request)

}

override def asyncUpdatePossiblySensitiveTweet(

request: AsyncUpdatePossiblySensitiveTweetRequest

): Future[Unit] =

track("async\_update\_possibly\_sensitive\_tweet", request) {

underlying.asyncUpdatePossiblySensitiveTweet(request)

}

override def replicatedUpdatePossiblySensitiveTweet(tweet: Tweet): Future[Unit] =

track("replicated\_update\_possibly\_sensitive\_tweet", tweet) {

underlying.replicatedUpdatePossiblySensitiveTweet(tweet)

}

override def undeleteTweet(request: UndeleteTweetRequest): Future[UndeleteTweetResponse] =

track("undelete\_tweet", request) {

underlying.undeleteTweet(request)

}

override def asyncUndeleteTweet(request: AsyncUndeleteTweetRequest): Future[Unit] =

track("async\_undelete\_tweet", request) {

underlying.asyncUndeleteTweet(request)

}

override def unretweet(request: UnretweetRequest): Future[UnretweetResult] =

track("unretweet", request) {

underlying.unretweet(request)

}

override def eraseUserTweets(request: EraseUserTweetsRequest): Future[Unit] =

track("erase\_user\_tweets", request) {

underlying.eraseUserTweets(request)

}

override def asyncEraseUserTweets(request: AsyncEraseUserTweetsRequest): Future[Unit] =

track("async\_erase\_user\_tweets", request) {

underlying.asyncEraseUserTweets(request)

}

override def asyncDelete(request: AsyncDeleteRequest): Future[Unit] =

track("async\_delete", request) { underlying.asyncDelete(request) }

override def deleteTweets(request: DeleteTweetsRequest): Future[Seq[DeleteTweetResult]] =

track("delete\_tweets", request, requestSize = Some(request.tweetIds.size)) {

requestSizeAuthorizer(request.tweetIds.size, clientId).flatMap { \_ =>

underlying.deleteTweets(request)

}

}

override def cascadedDeleteTweet(request: CascadedDeleteTweetRequest): Future[Unit] =

track("cascaded\_delete\_tweet", request) { underlying.cascadedDeleteTweet(request) }

override def replicatedDeleteTweet2(request: ReplicatedDeleteTweet2Request): Future[Unit] =

track("replicated\_delete\_tweet2", request) { underlying.replicatedDeleteTweet2(request) }

override def incrTweetFavCount(request: IncrTweetFavCountRequest): Future[Unit] =

track("incr\_tweet\_fav\_count", request) { underlying.incrTweetFavCount(request) }

override def asyncIncrFavCount(request: AsyncIncrFavCountRequest): Future[Unit] =

track("async\_incr\_fav\_count", request) { underlying.asyncIncrFavCount(request) }

override def replicatedIncrFavCount(tweetId: TweetId, delta: Int): Future[Unit] =

track("replicated\_incr\_fav\_count", tweetId) {

underlying.replicatedIncrFavCount(tweetId, delta)

}

override def incrTweetBookmarkCount(request: IncrTweetBookmarkCountRequest): Future[Unit] =

track("incr\_tweet\_bookmark\_count", request) { underlying.incrTweetBookmarkCount(request) }

override def asyncIncrBookmarkCount(request: AsyncIncrBookmarkCountRequest): Future[Unit] =

track("async\_incr\_bookmark\_count", request) { underlying.asyncIncrBookmarkCount(request) }

override def replicatedIncrBookmarkCount(tweetId: TweetId, delta: Int): Future[Unit] =

track("replicated\_incr\_bookmark\_count", tweetId) {

underlying.replicatedIncrBookmarkCount(tweetId, delta)

}

override def replicatedSetAdditionalFields(request: SetAdditionalFieldsRequest): Future[Unit] =

track("replicated\_set\_additional\_fields", request) {

underlying.replicatedSetAdditionalFields(request)

}

def setRetweetVisibility(request: SetRetweetVisibilityRequest): Future[Unit] = {

track("set\_retweet\_visibility", request) {

underlying.setRetweetVisibility(request)

}

}

def asyncSetRetweetVisibility(request: AsyncSetRetweetVisibilityRequest): Future[Unit] = {

track("async\_set\_retweet\_visibility", request) {

underlying.asyncSetRetweetVisibility(request)

}

}

override def replicatedSetRetweetVisibility(

request: ReplicatedSetRetweetVisibilityRequest

): Future[Unit] =

track("replicated\_set\_retweet\_visibility", request) {

underlying.replicatedSetRetweetVisibility(request)

}

override def replicatedDeleteAdditionalFields(

request: ReplicatedDeleteAdditionalFieldsRequest

): Future[Unit] =

track("replicated\_delete\_additional\_fields", request) {

underlying.replicatedDeleteAdditionalFields(request)

}

override def replicatedTakedown(tweet: Tweet): Future[Unit] =

track("replicated\_takedown", tweet) { underlying.replicatedTakedown(tweet) }

override def scrubGeoUpdateUserTimestamp(request: DeleteLocationData): Future[Unit] =

track("scrub\_geo\_update\_user\_timestamp", request) {

underlying.scrubGeoUpdateUserTimestamp(request)

}

override def scrubGeo(request: GeoScrub): Future[Unit] =

track("scrub\_geo", request, requestSize = Some(request.statusIds.size)) {

requestSizeAuthorizer(request.statusIds.size, clientId).flatMap { \_ =>

underlying.scrubGeo(request)

}

}

override def replicatedScrubGeo(tweetIds: Seq[TweetId]): Future[Unit] =

track("replicated\_scrub\_geo", tweetIds) { underlying.replicatedScrubGeo(tweetIds) }

override def deleteLocationData(request: DeleteLocationDataRequest): Future[Unit] =

track("delete\_location\_data", request) {

underlying.deleteLocationData(request)

}

override def flush(request: FlushRequest): Future[Unit] =

track("flush", request, requestSize = Some(request.tweetIds.size)) {

requestSizeAuthorizer(request.tweetIds.size, clientId).flatMap { \_ =>

underlying.flush(request)

}

}

override def takedown(request: TakedownRequest): Future[Unit] =

track("takedown", request) { underlying.takedown(request) }

override def asyncTakedown(request: AsyncTakedownRequest): Future[Unit] =

track("async\_takedown", request) {

underlying.asyncTakedown(request)

}

override def setTweetUserTakedown(request: SetTweetUserTakedownRequest): Future[Unit] =

track("set\_tweet\_user\_takedown", request) { underlying.setTweetUserTakedown(request) }

override def quotedTweetDelete(request: QuotedTweetDeleteRequest): Future[Unit] =

track("quoted\_tweet\_delete", request) {

underlying.quotedTweetDelete(request)

}

override def quotedTweetTakedown(request: QuotedTweetTakedownRequest): Future[Unit] =

track("quoted\_tweet\_takedown", request) {

underlying.quotedTweetTakedown(request)

}

override def getDeletedTweets(

request: GetDeletedTweetsRequest

): Future[Seq[GetDeletedTweetResult]] =

track("get\_deleted\_tweets", request, requestSize = Some(request.tweetIds.size)) {

requestSizeAuthorizer(request.tweetIds.size, clientId).flatMap { \_ =>

underlying.getDeletedTweets(request)

}

}

override def getStoredTweets(

request: GetStoredTweetsRequest

): Future[Seq[GetStoredTweetsResult]] = {

track("get\_stored\_tweets", request, requestSize = Some(request.tweetIds.size)) {

requestSizeAuthorizer(request.tweetIds.size, clientId).flatMap { \_ =>

underlying.getStoredTweets(request)

}

}

}

override def getStoredTweetsByUser(

request: GetStoredTweetsByUserRequest

): Future[GetStoredTweetsByUserResult] = {

track("get\_stored\_tweets\_by\_user", request) {

underlying.getStoredTweetsByUser(request)

}

}

}