package com.twitter.tweetypie.storage

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.mtls.authentication.EmptyServiceIdentifier

import com.twitter.finagle.mtls.authentication.ServiceIdentifier

import com.twitter.finagle.ssl.OpportunisticTls

import com.twitter.finagle.stats.NullStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.logging.BareFormatter

import com.twitter.logging.Level

import com.twitter.logging.ScribeHandler

import com.twitter.logging.\_

import com.twitter.stitch.Stitch

import com.twitter.storage.client.manhattan.bijections.Bijections.\_

import com.twitter.storage.client.manhattan.kv.\_

import com.twitter.storage.client.manhattan.kv.impl.ValueDescriptor

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

import com.twitter.tweetypie.storage.Scribe.ScribeHandlerFactory

import com.twitter.tweetypie.storage.TweetStorageClient.BounceDelete

import com.twitter.tweetypie.storage.TweetStorageClient.GetTweet

import com.twitter.tweetypie.storage.TweetStorageClient.HardDeleteTweet

import com.twitter.tweetypie.thriftscala.Tweet

import com.twitter.tweetypie.util.StitchUtils

import com.twitter.util.Duration

import com.twitter.util.Return

import com.twitter.util.Throw

import scala.util.Random

object ManhattanTweetStorageClient {

object Config {

/\*\*

\* The Manhattan dataset where tweets are stored is not externally

\* configurable because writing tweets to a non-production dataset

\* requires great care. Staging instances using a different dataset will

\* write tweets to a non-production store, but will publish events, log to

\* HDFS, and cache data referencing tweets in that store which are not

\* accessible by the rest of the production cluster.

\*

\* In a completely isolated environment it should be safe to write to

\* other datasets for testing purposes.

\*/

val Dataset = "tbird\_mh"

/\*\*

\* Once a tweet has been deleted it can only be undeleted within this time

\* window, after which [[UndeleteHandler]] will return an error on

\* undelete attempts.

\*/

val UndeleteWindowHours = 240

/\*\*

\* Default label used for underlying Manhattan Thrift client metrics

\*

\* The finagle client metrics will be exported at clnt/:label.

\*/

val ThriftClientLabel = "mh\_cylon"

/\*\*

\* Return the corresponding Wily path for the Cylon cluster in the "other" DC

\*/

def remoteDestination(zone: String): String =

s"/srv#/prod/${remoteZone(zone)}/manhattan/cylon.native-thrift"

private def remoteZone(zone: String) = zone match {

case "pdxa" => "atla"

case "atla" | "localhost" => "pdxa"

case \_ =>

throw new IllegalArgumentException(s"Cannot configure remote DC for unknown zone '$zone'")

}

}

/\*\*

\* @param applicationId Manhattan application id used for quota accounting

\* @param localDestination Wily path to local Manhattan cluster

\* @param localTimeout Overall timeout (including retries) for all reads/writes to local cluster

\* @param remoteDestination Wily path to remote Manhattan cluster, used for undelete and force add

\* @param remoteTimeout Overall timeout (including retries) for all reads/writes to remote cluster

\* @param undeleteWindowHours Amount of time during which a deleted tweet can be undeleted

\* @param thriftClientLabel Label used to scope stats for Manhattan Thrift client

\* @param maxRequestsPerBatch Configure the Stitch RequestGroup.Generator batch size

\* @param serviceIdentifier The ServiceIdentifier to use when making connections to a Manhattan cluster

\* @param opportunisticTlsLevel The level to use for opportunistic TLS for connections to the Manhattan cluster

\*/

case class Config(

applicationId: String,

localDestination: String,

localTimeout: Duration,

remoteDestination: String,

remoteTimeout: Duration,

undeleteWindowHours: Int = Config.UndeleteWindowHours,

thriftClientLabel: String = Config.ThriftClientLabel,

maxRequestsPerBatch: Int = Int.MaxValue,

serviceIdentifier: ServiceIdentifier,

opportunisticTlsLevel: OpportunisticTls.Level)

/\*\*

\* Sanitizes the input for APIs which take in a (Tweet, Seq[Field]) as input.

\*

\* NOTE: This function only applies sanity checks which are common to

\* all APIs which take in a (Tweet, Seq[Field]) as input. API specific

\* checks are not covered here.

\*

\* @param apiStitch the backing API call

\* @tparam T the output type of the backing API call

\* @return a stitch function which does some basic input sanity checking

\*/

private[storage] def sanitizeTweetFields[T](

apiStitch: (Tweet, Seq[Field]) => Stitch[T]

): (Tweet, Seq[Field]) => Stitch[T] =

(tweet, fields) => {

require(fields.forall(\_.id > 0), s"Field ids ${fields} are not positive numbers")

apiStitch(tweet, fields)

}

// Returns a handler that asynchronously logs messages to Scribe using the BareFormatter which

// logs just the message without any additional metadata

def scribeHandler(categoryName: String): HandlerFactory =

ScribeHandler(

formatter = BareFormatter,

maxMessagesPerTransaction = 100,

category = categoryName,

level = Some(Level.TRACE)

)

/\*\*

\* A Config appropriate for interactive sessions and scripts.

\*/

def develConfig(): Config =

Config(

applicationId = Option(System.getenv("USER")).getOrElse("<unknown>") + ".devel",

localDestination = "/s/manhattan/cylon.native-thrift",

localTimeout = 10.seconds,

remoteDestination = "/s/manhattan/cylon.native-thrift",

remoteTimeout = 10.seconds,

undeleteWindowHours = Config.UndeleteWindowHours,

thriftClientLabel = Config.ThriftClientLabel,

maxRequestsPerBatch = Int.MaxValue,

serviceIdentifier = ServiceIdentifier(System.getenv("USER"), "tweetypie", "devel", "local"),

opportunisticTlsLevel = OpportunisticTls.Required

)

/\*\*

\* Build a Manhattan tweet storage client for use in interactive

\* sessions and scripts.

\*/

def devel(): TweetStorageClient =

new ManhattanTweetStorageClient(

develConfig(),

NullStatsReceiver,

ClientIdHelper.default,

)

}

class ManhattanTweetStorageClient(

config: ManhattanTweetStorageClient.Config,

statsReceiver: StatsReceiver,

private val clientIdHelper: ClientIdHelper)

extends TweetStorageClient {

import ManhattanTweetStorageClient.\_

lazy val scribeHandlerFactory: ScribeHandlerFactory = scribeHandler \_

val scribe: Scribe = new Scribe(scribeHandlerFactory, statsReceiver)

def mkClient(

dest: String,

label: String

): ManhattanKVClient = {

val mhMtlsParams =

if (config.serviceIdentifier == EmptyServiceIdentifier) NoMtlsParams

else

ManhattanKVClientMtlsParams(

serviceIdentifier = config.serviceIdentifier,

opportunisticTls = config.opportunisticTlsLevel

)

new ManhattanKVClient(

config.applicationId,

dest,

mhMtlsParams,

label,

Seq(Experiments.ApertureLoadBalancer))

}

val localClient: ManhattanKVClient = mkClient(config.localDestination, config.thriftClientLabel)

val localMhEndpoint: ManhattanKVEndpoint = ManhattanKVEndpointBuilder(localClient)

.defaultGuarantee(Guarantee.SoftDcReadMyWrites)

.defaultMaxTimeout(config.localTimeout)

.maxRequestsPerBatch(config.maxRequestsPerBatch)

.build()

val localManhattanOperations = new ManhattanOperations(Config.Dataset, localMhEndpoint)

val remoteClient: ManhattanKVClient =

mkClient(config.remoteDestination, s"${config.thriftClientLabel}\_remote")

val remoteMhEndpoint: ManhattanKVEndpoint = ManhattanKVEndpointBuilder(remoteClient)

.defaultGuarantee(Guarantee.SoftDcReadMyWrites)

.defaultMaxTimeout(config.remoteTimeout)

.build()

val remoteManhattanOperations = new ManhattanOperations(Config.Dataset, remoteMhEndpoint)

/\*\*

\* Note: This translation is only useful for non-batch endpoints. Batch endpoints currently

\* represent failure without propagating an exception

\* (e.g. [[com.twitter.tweetypie.storage.Response.TweetResponseCode.Failure]]).

\*/

private[this] def translateExceptions(

apiName: String,

statsReceiver: StatsReceiver

): PartialFunction[Throwable, Throwable] = {

case e: IllegalArgumentException => ClientError(e.getMessage, e)

case e: DeniedManhattanException => RateLimited(e.getMessage, e)

case e: VersionMismatchError =>

statsReceiver.scope(apiName).counter("mh\_version\_mismatches").incr()

e

case e: InternalError =>

TweetUtils.log.error(e, s"Error processing $apiName request: ${e.getMessage}")

e

}

/\*\*

\* Count requests per client id producing metrics of the form

\* .../clients/:root\_client\_id/requests

\*/

def observeClientId[A, B](

apiStitch: A => Stitch[B],

statsReceiver: StatsReceiver,

clientIdHelper: ClientIdHelper,

): A => Stitch[B] = {

val clients = statsReceiver.scope("clients")

val incrementClientRequests = { args: A =>

val clientId = clientIdHelper.effectiveClientIdRoot.getOrElse(ClientIdHelper.UnknownClientId)

clients.counter(clientId, "requests").incr

}

a => {

incrementClientRequests(a)

apiStitch(a)

}

}

/\*\*

\* Increment counters based on the overall response status of the returned [[GetTweet.Response]].

\*/

def observeGetTweetResponseCode[A](

apiStitch: A => Stitch[GetTweet.Response],

statsReceiver: StatsReceiver

): A => Stitch[GetTweet.Response] = {

val scope = statsReceiver.scope("response\_code")

val success = scope.counter("success")

val notFound = scope.counter("not\_found")

val failure = scope.counter("failure")

val overCapacity = scope.counter("over\_capacity")

val deleted = scope.counter("deleted")

val bounceDeleted = scope.counter("bounce\_deleted")

a =>

apiStitch(a).respond {

case Return(\_: GetTweet.Response.Found) => success.incr()

case Return(GetTweet.Response.NotFound) => notFound.incr()

case Return(\_: GetTweet.Response.BounceDeleted) => bounceDeleted.incr()

case Return(GetTweet.Response.Deleted) => deleted.incr()

case Throw(\_: RateLimited) => overCapacity.incr()

case Throw(\_) => failure.incr()

}

}

/\*\*

\* We do 3 things here:

\*

\* - Bookkeeping for overall requests

\* - Bookkeeping for per api requests

\* - Translate exceptions

\*

\* @param apiName the API being called

\* @param apiStitch the implementation of the API

\* @tparam A template for input type of API

\* @tparam B template for output type of API

\* @return Function which executes the given API call

\*/

private[storage] def endpoint[A, B](

apiName: String,

apiStitch: A => Stitch[B]

): A => Stitch[B] = {

val translateException = translateExceptions(apiName, statsReceiver)

val observe = StitchUtils.observe[B](statsReceiver, apiName)

a =>

StitchUtils.translateExceptions(

observe(apiStitch(a)),

translateException

)

}

private[storage] def endpoint2[A, B, C](

apiName: String,

apiStitch: (A, B) => Stitch[C],

clientIdHelper: ClientIdHelper,

): (A, B) => Stitch[C] =

Function.untupled(endpoint(apiName, apiStitch.tupled))

val getTweet: TweetStorageClient.GetTweet = {

val stats = statsReceiver.scope("getTweet")

observeClientId(

observeGetTweetResponseCode(

endpoint(

"getTweet",

GetTweetHandler(

read = localManhattanOperations.read,

statsReceiver = stats,

)

),

stats,

),

stats,

clientIdHelper,

)

}

val getStoredTweet: TweetStorageClient.GetStoredTweet = {

val stats = statsReceiver.scope("getStoredTweet")

observeClientId(

endpoint(

"getStoredTweet",

GetStoredTweetHandler(

read = localManhattanOperations.read,

statsReceiver = stats,

)

),

stats,

clientIdHelper,

)

}

val addTweet: TweetStorageClient.AddTweet =

endpoint(

"addTweet",

AddTweetHandler(

insert = localManhattanOperations.insert,

scribe = scribe,

stats = statsReceiver

)

)

val updateTweet: TweetStorageClient.UpdateTweet =

endpoint2(

"updateTweet",

ManhattanTweetStorageClient.sanitizeTweetFields(

UpdateTweetHandler(

insert = localManhattanOperations.insert,

stats = statsReceiver,

)

),

clientIdHelper,

)

val softDelete: TweetStorageClient.SoftDelete =

endpoint(

"softDelete",

SoftDeleteHandler(

insert = localManhattanOperations.insert,

scribe = scribe

)

)

val bounceDelete: BounceDelete =

endpoint(

"bounceDelete",

BounceDeleteHandler(

insert = localManhattanOperations.insert,

scribe = scribe

)

)

val undelete: TweetStorageClient.Undelete =

endpoint(

"undelete",

UndeleteHandler(

read = localManhattanOperations.read,

localInsert = localManhattanOperations.insert,

remoteInsert = remoteManhattanOperations.insert,

delete = localManhattanOperations.delete,

undeleteWindowHours = config.undeleteWindowHours,

stats = statsReceiver

)

)

val getDeletedTweets: TweetStorageClient.GetDeletedTweets =

endpoint(

"getDeletedTweets",

GetDeletedTweetsHandler(

read = localManhattanOperations.read,

stats = statsReceiver

)

)

val deleteAdditionalFields: TweetStorageClient.DeleteAdditionalFields =

endpoint2(

"deleteAdditionalFields",

DeleteAdditionalFieldsHandler(

delete = localManhattanOperations.delete,

stats = statsReceiver,

),

clientIdHelper,

)

val scrub: TweetStorageClient.Scrub =

endpoint2(

"scrub",

ScrubHandler(

insert = localManhattanOperations.insert,

delete = localManhattanOperations.delete,

scribe = scribe,

stats = statsReceiver,

),

clientIdHelper,

)

val hardDeleteTweet: HardDeleteTweet =

endpoint(

"hardDeleteTweet",

HardDeleteTweetHandler(

read = localManhattanOperations.read,

insert = localManhattanOperations.insert,

delete = localManhattanOperations.delete,

scribe = scribe,

stats = statsReceiver

)

)

val ping: TweetStorageClient.Ping =

() =>

Stitch

.run(

localMhEndpoint

.get(

ManhattanOperations.KeyDescriptor

.withDataset(Config.Dataset)

.withPkey(Random.nextLong().abs)

.withLkey(TweetKey.LKey.CoreFieldsKey), // could be any lkey

ValueDescriptor(BufInjection)

).unit

)

}