package com.twitter.usersignalservice

package base

import com.twitter.finagle.memcached.{Client => MemcachedClient}

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.hashing.KeyHasher

import com.twitter.hermit.store.common.ObservedMemcachedReadableStore

import com.twitter.relevance\_platform.common.injection.LZ4Injection

import com.twitter.relevance\_platform.common.injection.SeqObjectInjection

import com.twitter.storehaus.ReadableStore

import com.twitter.twistly.common.UserId

import com.twitter.usersignalservice.thriftscala.Signal

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.Timer

/\*\*

\* Use this wrapper when the latency of the signal fetcher is too high (see BaseSignalFetcher.Timeout

\* ) and the results from the signal fetcher don't change often (e.g. results are generated from a

\* scalding job scheduled each day).

\* @param memcachedClient

\* @param baseSignalFetcher

\* @param ttl

\* @param stats

\* @param timer

\*/

case class MemcachedSignalFetcherWrapper(

memcachedClient: MemcachedClient,

baseSignalFetcher: BaseSignalFetcher,

ttl: Duration,

stats: StatsReceiver,

keyPrefix: String,

timer: Timer)

extends BaseSignalFetcher {

import MemcachedSignalFetcherWrapper.\_

override type RawSignalType = baseSignalFetcher.RawSignalType

override val name: String = this.getClass.getCanonicalName

override val statsReceiver: StatsReceiver = stats.scope(name).scope(baseSignalFetcher.name)

val underlyingStore: ReadableStore[UserId, Seq[RawSignalType]] = {

val cacheUnderlyingStore = new ReadableStore[UserId, Seq[RawSignalType]] {

override def get(userId: UserId): Future[Option[Seq[RawSignalType]]] =

baseSignalFetcher.getRawSignals(userId)

}

ObservedMemcachedReadableStore.fromCacheClient(

backingStore = cacheUnderlyingStore,

cacheClient = memcachedClient,

ttl = ttl)(

valueInjection = LZ4Injection.compose(SeqObjectInjection[RawSignalType]()),

statsReceiver = statsReceiver,

keyToString = { k: UserId =>

s"$keyPrefix:${keyHasher.hashKey(k.toString.getBytes)}"

}

)

}

override def getRawSignals(userId: UserId): Future[Option[Seq[RawSignalType]]] =

underlyingStore.get(userId)

override def process(

query: Query,

rawSignals: Future[Option[Seq[RawSignalType]]]

): Future[Option[Seq[Signal]]] = baseSignalFetcher.process(query, rawSignals)

}

object MemcachedSignalFetcherWrapper {

private val keyHasher: KeyHasher = KeyHasher.FNV1A\_64

}