package com.twitter.servo.repository

import com.twitter.servo.cache.\_

import com.twitter.util.\_

object ResponseCachingKeyValueRepository {

/\*\*

\* An cache filter that excludes cached future responses that are already fulfilled.

\* Using this policy ensures that this repository will only ever have one outstanding request for the same item.

\*/

def refreshSatisfied[K, V]: (K, Future[Option[V]]) => Boolean =

(\_, v) => v.isDefined

/\*\*

\* An cache filter that excludes cached future response that are failures

\*/

def refreshFailures[K, V]: (K, Future[Option[V]]) => Boolean =

(\_, v) =>

v.poll match {

case Some(t) => t.isThrow

case None => false

}

}

/\*\*

\* A repository that caches(in-process) Future responses from an underlying KeyValueRepository.

\* Each time a request for a key is made, the repository first checks

\* if any Future responses for that key are already cached.

\* If so, the Future response from cache is returned.

\* If not, a new Promise is placed in to cache,

\* the underlying repository is queried to fulfill the Promise,

\* and the new Promise is returned to the caller.

\* @param underlying

\* the underlying KeyValueRepository

\* @param cache

\* an inprocess cache of (future) responses

\* @param newQuery

\* a function which constructs a new query from a query and a set of keys

\* @param observer

\* a CacheObserver which records the hits/misses on the request cache

\*/

class ResponseCachingKeyValueRepository[Q <: Seq[K], K, V](

underlying: KeyValueRepository[Q, K, V],

cache: InProcessCache[K, Future[Option[V]]],

newQuery: SubqueryBuilder[Q, K],

observer: CacheObserver = NullCacheObserver)

extends KeyValueRepository[Q, K, V] {

private[this] def load(query: Q, promises: Seq[(K, Promise[Option[V]])]): Unit = {

if (promises.nonEmpty) {

underlying(newQuery(promises map { case (k, \_) => k }, query)) respond {

case Throw(t) => promises foreach { case (\_, p) => p.updateIfEmpty(Throw(t)) }

case Return(kvr) => promises foreach { case (k, p) => p.updateIfEmpty(kvr(k)) }

}

}

}

sealed trait RefreshResult[K, V] {

def toInterruptible: Future[Option[V]]

}

private case class CachedResult[K, V](result: Future[Option[V]]) extends RefreshResult[K, V] {

def toInterruptible = result.interruptible

}

private case class LoadResult[K, V](keyToLoad: K, result: Promise[Option[V]])

extends RefreshResult[K, V] {

def toInterruptible = result.interruptible

}

private[this] def refresh(key: K): RefreshResult[K, V] =

synchronized {

cache.get(key) match {

case Some(updated) =>

observer.hit(key.toString)

CachedResult(updated)

case None =>

observer.miss(key.toString)

val promise = new Promise[Option[V]]

cache.set(key, promise)

LoadResult(key, promise)

}

}

def apply(query: Q): Future[KeyValueResult[K, V]] =

KeyValueResult.fromSeqFuture(query) {

val result: Seq[RefreshResult[K, V]] =

query map { key =>

cache.get(key) match {

case Some(value) =>

observer.hit(key.toString)

CachedResult[K, V](value)

case None =>

refresh(key)

}

}

val toLoad = result collect { case LoadResult(k, p) => k -> p }

load(query, toLoad)

result map { \_.toInterruptible }

}

}