package com.twitter.tweetypie

package repository

import com.twitter.servo.repository.\_

import com.twitter.stitch.Stitch

import com.twitter.util.Try

object CacheStitch {

/\*\*

\* Cacheable defines a function that takes a cache query and a Try value,

\* and returns what should be written to cache, as a Option[StitchLockingCache.Val].

\*

\* None signifies that this value should not be written to cache.

\*

\* Val can be one of Found[V], NotFound, and Deleted. The function will determine what kinds

\* of values and exceptions (captured in the Try) correspond to which kind of cached values.

\*/

type Cacheable[Q, V] = (Q, Try[V]) => Option[StitchLockingCache.Val[V]]

// Cache successful values as Found, stitch.NotFound as NotFound, and don't cache other exceptions

def cacheFoundAndNotFound[K, V]: CacheStitch.Cacheable[K, V] =

(\_, t: Try[V]) =>

t match {

// Write successful values as Found

case Return(v) => Some(StitchLockingCache.Val.Found[V](v))

// Write stitch.NotFound as NotFound

case Throw(com.twitter.stitch.NotFound) => Some(StitchLockingCache.Val.NotFound)

// Don't write other exceptions back to cache

case \_ => None

}

}

case class CacheStitch[Q, K, V](

repo: Q => Stitch[V],

cache: StitchLockingCache[K, V],

queryToKey: Q => K,

handler: CachedResult.Handler[K, V],

cacheable: CacheStitch.Cacheable[Q, V])

extends (Q => Stitch[V]) {

import com.twitter.servo.repository.CachedResultAction.\_

private[this] def getFromCache(key: K): Stitch[CachedResult[K, V]] = {

cache

.get(key)

.handle {

case t => CachedResult.Failed(key, t)

}

}

// Exposed for testing

private[repository] def readThrough(query: Q): Stitch[V] =

repo(query).liftToTry.applyEffect { value: Try[V] =>

cacheable(query, value) match {

case Some(v) =>

// cacheable returned Some of a StitchLockingCache.Val to cache

//

// This is async to ensure that we don't wait for the cache

// update to complete before returning. This also ignores

// any exceptions from setting the value.

Stitch.async(cache.lockAndSet(queryToKey(query), v))

case None =>

// cacheable returned None so don't cache

Stitch.Unit

}

}.lowerFromTry

private[this] def handle(query: Q, action: CachedResultAction[V]): Stitch[V] =

action match {

case HandleAsFound(value) => Stitch(value)

case HandleAsMiss => readThrough(query)

case HandleAsDoNotCache => repo(query)

case HandleAsFailed(t) => Stitch.exception(t)

case HandleAsNotFound => Stitch.NotFound

case t: TransformSubAction[V] => handle(query, t.action).map(t.f)

case SoftExpiration(subAction) =>

Stitch

.async(readThrough(query))

.flatMap { \_ => handle(query, subAction) }

}

override def apply(query: Q): Stitch[V] =

getFromCache(queryToKey(query))

.flatMap { result: CachedResult[K, V] => handle(query, handler(result)) }

}