package com.twitter.tweetypie.caching

import com.twitter.stitch.MapGroup

import com.twitter.stitch.Group

import com.twitter.stitch.Stitch

import com.twitter.util.Future

import com.twitter.util.Return

import com.twitter.util.Try

/\*\*

\* Wrapper around [[CacheOperations]] providing a [[Stitch]] API.

\*/

case class StitchCacheOperations[K, V](operations: CacheOperations[K, V]) {

import StitchCacheOperations.SetCall

private[this] val getGroup: Group[K, CacheResult[V]] =

MapGroup[K, CacheResult[V]] { keys: Seq[K] =>

operations

.get(keys)

.map(values => keys.zip(values).toMap.mapValues(Return(\_)))

}

def get(key: K): Stitch[CacheResult[V]] =

Stitch.call(key, getGroup)

private[this] val setGroup: Group[SetCall[K, V], Unit] =

new MapGroup[SetCall[K, V], Unit] {

override def run(calls: Seq[SetCall[K, V]]): Future[SetCall[K, V] => Try[Unit]] =

Future

.collectToTry(calls.map(call => operations.set(call.key, call.value)))

.map(tries => calls.zip(tries).toMap)

}

/\*\*

\* Performs a [[CacheOperations.set]].

\*/

def set(key: K, value: V): Stitch[Unit] =

// This is implemented as a Stitch.call instead of a Stitch.future

// in order to handle the case where a batch has a duplicate

// key. Each copy of the duplicate key will trigger a write back

// to cache, so we dedupe the writes in order to avoid the

// extraneous RPC call.

Stitch.call(new StitchCacheOperations.SetCall(key, value), setGroup)

}

object StitchCacheOperations {

/\*\*

\* Used as the "call" for [[SetGroup]]. This is essentially a tuple

\* where equality is defined only by the key.

\*/

private class SetCall[K, V](val key: K, val value: V) {

override def equals(other: Any): Boolean =

other match {

case setCall: SetCall[\_, \_] => key == setCall.key

case \_ => false

}

override def hashCode: Int = key.hashCode

}

}