package com.twitter.servo.store

import com.twitter.servo.cache.{Cached, CachedValueStatus, LockingCache}

import com.twitter.logging.Logger

import com.twitter.util.{Future, Time}

/\*\*

\* Wraps a cache around an underlying store.

\*

\* CachingStore is a specialization of TransformingCachingStore where the store and cache are

\* assumed to have the same key and value types. See TransformingCachingStore for a discussion

\* of the arguments to CachingStore.

\*/

class CachingStore[K, V](

cache: LockingCache[K, Cached[V]],

underlying: Store[K, V],

valuePicker: LockingCache.Picker[Cached[V]],

key: V => K)

extends TransformingCachingStore[K, V, K, V](

cache,

underlying,

valuePicker,

key,

identity,

identity

)

/\*\*

\* Wraps a cache of differing key/value types around an underlying store.

\*

\* Updates are applied first (unmodified) to the underlying store and then

\* the cache is updated after running the key/value through a one-way function

\* to derive the key/value as expected by the cache.

\*

\* @param cache

\* the wrapping cache

\*

\* @param underlying

\* the underlying store

\*

\* @param valuePicker

\* chooses between existing and new value

\*

\* @param key

\* computes a key from the value being stored

\*

\* @param cacheKey

\* transforms the store's key type to the cache's key type

\*

\* @param cacheValue

\* transforms the store's value type to the cache's value type

\*/

class TransformingCachingStore[K, V, CacheK, CacheV](

cache: LockingCache[CacheK, Cached[CacheV]],

underlying: Store[K, V],

valuePicker: LockingCache.Picker[Cached[CacheV]],

key: V => K,

cacheKey: K => CacheK,

cacheValue: V => CacheV)

extends Store[K, V] {

protected[this] val log = Logger.get(getClass.getSimpleName)

override def create(value: V): Future[V] = {

chainCacheOp[V](

underlying.create(value),

result => cache(key(result), Some(result), CachedValueStatus.Found, "new")

)

}

override def update(value: V): Future[Unit] = {

chainCacheOp[Unit](

underlying.update(value),

\_ => cache(key(value), Some(value), CachedValueStatus.Found, "updated")

)

}

override def destroy(key: K): Future[Unit] = {

chainCacheOp[Unit](

underlying.destroy(key),

\_ => cache(key, None, CachedValueStatus.Deleted, "deleted")

)

}

/\*\*

\* Subclasses may override this to alter the relationship between the result

\* of the underlying Store operation and the result of the Cache operation.

\* By default, the cache operation occurs asynchronously and only upon success

\* of the store operation. Cache operation failures are logged but otherwise

\* ignored.

\*/

protected[this] def chainCacheOp[Result](

storeOp: Future[Result],

cacheOp: Result => Future[Unit]

): Future[Result] = {

storeOp onSuccess { cacheOp(\_) }

}

protected[this] def cache(

key: K,

value: Option[V],

status: CachedValueStatus,

desc: String

): Future[Unit] = {

val now = Time.now

val cached = Cached(value map { cacheValue(\_) }, status, now, None, Some(now))

val handler = LockingCache.PickingHandler(cached, valuePicker)

cache.lockAndSet(cacheKey(key), handler).unit onFailure {

case t =>

log.error(t, "exception caught while caching %s value", desc)

}

}

}