package com.twitter.servo.hydrator

import com.twitter.servo.data.Mutation

import com.twitter.servo.util.{Effect, Gate}

import com.twitter.servo.repository.\_

import com.twitter.util.{Future, Return, Try}

object KeyValueHydrator {

// KeyValueHydrator extends this function type

type FunctionType[Q, K, V] = (Q, Future[KeyValueResult[K, V]]) => Future[Mutation[V]]

type Filter[Q, K, V] = (Q, Future[KeyValueResult[K, V]]) => Future[Boolean]

private[this] val \_unit = fromMutation[Any, Any, Any](Mutation.unit[Any])

/\*\*

\* A no-op hydrator. Forms a monoid with `also`.

\*/

def unit[Q, K, V]: KeyValueHydrator[Q, K, V] =

\_unit.asInstanceOf[KeyValueHydrator[Q, K, V]]

/\*\*

\* Packages a function as a KeyValueHydrator

\*/

def apply[Q, K, V](f: FunctionType[Q, K, V]): KeyValueHydrator[Q, K, V] =

new KeyValueHydrator[Q, K, V] {

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

f(query, futureResults)

}

/\*\*

\* Creates a new KeyValueHydrator out of several underlying KVHydrators. The

\* apply method is called on each KeyValueHydrator with the same

\* futureResults, allowing each to kick-off some asynchronous work

\* to produce a future Hydrated[Mutation]. When all the future

\* Hydrated[Mutation]s are available, the results are folded,

\* left-to-right, over the mutations, to build up the final

\* results.

\*/

def inParallel[Q, K, V](hydrators: KeyValueHydrator[Q, K, V]\*): KeyValueHydrator[Q, K, V] =

KeyValueHydrator[Q, K, V] { (query, futureResults) =>

val futureMutations = hydrators map { t =>

t(query, futureResults)

}

Future.collect(futureMutations) map Mutation.all

}

def const[Q, K, V](futureMutation: Future[Mutation[V]]): KeyValueHydrator[Q, K, V] =

KeyValueHydrator[Q, K, V] { (\_, \_) =>

futureMutation

}

def fromMutation[Q, K, V](mutation: Mutation[V]): KeyValueHydrator[Q, K, V] =

const[Q, K, V](Future.value(mutation))

}

/\*\*

\* A KeyValueHydrator builds a Mutation to be applied to the values in a KeyValueResult, but does

\* not itself apply the Mutation. This allows several KeyValueHydrators to be composed together to

\* begin their work in parallel to build the Mutations, which can then be combined and applied

\* to the results later (see asRepositoryFilter).

\*

\* Forms a monoid with KeyValueHydrator.unit as unit and `also` as the combining function.

\*/

trait KeyValueHydrator[Q, K, V] extends KeyValueHydrator.FunctionType[Q, K, V] {

protected[this] val unitMutation = Mutation.unit[V]

protected[this] val futureUnitMutation = Future.value(unitMutation)

/\*\*

\* Combines two KeyValueHydrators. Forms a monoid with KeyValueHydator.unit

\*/

def also(next: KeyValueHydrator[Q, K, V]): KeyValueHydrator[Q, K, V] =

KeyValueHydrator.inParallel(this, next)

/\*\*

\* Turns a single KeyValueHydrator into a RepositoryFilter by applying the Mutation to

\* found values in the KeyValueResult. If the mutation throws an exception, it will

\* be caught and the resulting key/value paired moved to the failed map of the resulting

\* KeyValueResult.

\*/

lazy val asRepositoryFilter: RepositoryFilter[Q, KeyValueResult[K, V], KeyValueResult[K, V]] =

(query, futureResults) => {

this(query, futureResults) flatMap { mutation =>

val update = mutation.endo

futureResults map { results =>

results.mapValues {

case Return(Some(value)) => Try(Some(update(value)))

case x => x

}

}

}

}

/\*\*

\* Apply this hydrator to the result of a repository.

\*/

def hydratedBy\_:(repo: KeyValueRepository[Q, K, V]): KeyValueRepository[Q, K, V] =

Repository.composed(repo, asRepositoryFilter)

/\*\*

\* Return a new hydrator that applies the same mutation as this

\* hydrator, but can be enabled/disabled or dark enabled/disabled via Gates. The light

\* gate takes precedence over the dark gate. This allows you to go from 0%->100% dark,

\* and then from 0%->100% light without affecting backend traffic.

\*/

@deprecated("Use enabledBy(() => Boolean, () => Boolean)", "2.5.1")

def enabledBy(light: Gate[Unit], dark: Gate[Unit] = Gate.False): KeyValueHydrator[Q, K, V] =

enabledBy(

{ () =>

light()

},

{ () =>

dark()

})

/\*\*

\* Return a new hydrator that applies the same mutation as this

\* hydrator, but can be enabled/disabled or dark enable/disabled via nullary boolean functions.

\* The light function takes precedence over the dark function.

\* This allows you to go from 0%->100% dark, and then from 0%->100% light

\* without affecting backend traffic.

\*/

def enabledBy(light: () => Boolean, dark: () => Boolean): KeyValueHydrator[Q, K, V] =

KeyValueHydrator[Q, K, V] { (query, futureResults) =>

val isLight = light()

val isDark = !isLight && dark()

if (!isLight && !isDark) {

futureUnitMutation

} else {

this(query, futureResults) map {

case mutation if isLight => mutation

case mutation if isDark => mutation.dark

}

}

}

/\*\*

\* Build a new hydrator that will return the same result as the current hydrator,

\* but will additionally perform the supplied effect on the result of hydration.

\*/

def withEffect(effect: Effect[Option[V]]): KeyValueHydrator[Q, K, V] =

KeyValueHydrator[Q, K, V] { (query, futureResults) =>

this(query, futureResults) map { \_ withEffect effect }

}

/\*\*

\* Builds a new hydrator that only attempt to hydrate if the

\* supplied filter returns true.

\*/

def filter(predicate: KeyValueHydrator.Filter[Q, K, V]): KeyValueHydrator[Q, K, V] =

KeyValueHydrator[Q, K, V] { (q, r) =>

predicate(q, r) flatMap { t =>

if (t) this(q, r) else futureUnitMutation

}

}

}