package com.twitter.tweetypie

package hydrator

import com.twitter.servo.util.ExceptionCounter

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.core.EditState

import com.twitter.tweetypie.core.ValueState

import com.twitter.util.Try

/\*\*

\* A ValueHydrator hydrates a value of type `A`, with a hydration context of type `C`,

\* and produces a value of type ValueState[A] (ValueState encapsulates the value and

\* its associated HydrationState).

\*

\* Because ValueHydrators take a value and produce a new value, they can easily be run

\* in sequence, but not in parallel. To run hydrators in parallel, see [[EditHydrator]].

\*

\* A series of ValueHydrators of the same type may be run in sequence via

\* `ValueHydrator.inSequence`.

\*

\*/

class ValueHydrator[A, C] private (val run: (A, C) => Stitch[ValueState[A]]) {

/\*\*

\* Apply this hydrator to a value, producing a ValueState.

\*/

def apply(a: A, ctx: C): Stitch[ValueState[A]] = run(a, ctx)

/\*\*

\* Apply with an empty context: only used in tests.

\*/

def apply(a: A)(implicit ev: Unit <:< C): Stitch[ValueState[A]] =

apply(a, ev(()))

/\*\*

\* Convert this ValueHydrator to the equivalent EditHydrator.

\*/

def toEditHydrator: EditHydrator[A, C] =

EditHydrator[A, C] { (a, ctx) => this.run(a, ctx).map(value => EditState(\_ => value)) }

/\*\*

\* Chains two ValueHydrators in sequence.

\*/

def andThen(next: ValueHydrator[A, C]): ValueHydrator[A, C] =

ValueHydrator[A, C] { (x0, ctx) =>

for {

r1 <- run(x0, ctx)

r2 <- next.run(r1.value, ctx)

} yield {

ValueState(r2.value, r1.state ++ r2.state)

}

}

/\*\*

\* Executes this ValueHydrator conditionally based on a Gate.

\*/

def ifEnabled(gate: Gate[Unit]): ValueHydrator[A, C] =

onlyIf((\_, \_) => gate())

/\*\*

\* Executes this ValueHydrator conditionally based on a boolean function.

\*/

def onlyIf(cond: (A, C) => Boolean): ValueHydrator[A, C] =

ValueHydrator { (a, c) =>

if (cond(a, c)) {

run(a, c)

} else {

Stitch.value(ValueState.unit(a))

}

}

/\*\*

\* Converts a ValueHydrator of input type `A` to input type `Option[A]`.

\*/

def liftOption: ValueHydrator[Option[A], C] =

liftOption(None)

/\*\*

\* Converts a ValueHydrator of input type `A` to input type `Option[A]` with a

\* default input value.

\*/

def liftOption(default: A): ValueHydrator[Option[A], C] =

liftOption(Some(default))

private def liftOption(default: Option[A]): ValueHydrator[Option[A], C] = {

val none = Stitch.value(ValueState.unit(None))

ValueHydrator[Option[A], C] { (a, ctx) =>

a.orElse(default) match {

case Some(a) => this.run(a, ctx).map(s => s.map(Some.apply))

case None => none

}

}

}

/\*\*

\* Converts a ValueHydrator of input type `A` to input type `Seq[A]`.

\*/

def liftSeq: ValueHydrator[Seq[A], C] =

ValueHydrator[Seq[A], C] { (as, ctx) =>

Stitch.traverse(as)(a => run(a, ctx)).map(rs => ValueState.sequence[A](rs))

}

/\*\*

\* Produces a new ValueHydrator that collects stats on the hydration.

\*/

def observe(

stats: StatsReceiver,

mkExceptionCounter: (StatsReceiver, String) => ExceptionCounter = (stats, scope) =>

new ExceptionCounter(stats, scope)

): ValueHydrator[A, C] = {

val callCounter = stats.counter("calls")

val noopCounter = stats.counter("noop")

val modifiedCounter = stats.counter("modified")

val partialCounter = stats.counter("partial")

val completedCounter = stats.counter("completed")

val exceptionCounter = mkExceptionCounter(stats, "failures")

ValueHydrator[A, C] { (a, ctx) =>

this.run(a, ctx).respond {

case Return(ValueState(\_, state)) =>

callCounter.incr()

if (state.isEmpty) {

noopCounter.incr()

} else {

if (state.modified) modifiedCounter.incr()

if (state.failedFields.nonEmpty) partialCounter.incr()

if (state.completedHydrations.nonEmpty) completedCounter.incr()

}

case Throw(ex) =>

callCounter.incr()

exceptionCounter(ex)

}

}

}

/\*\*

\* Produces a new ValueHydrator that uses a lens to extract the value to hydrate,

\* using this hydrator, and then to put the updated value back in the enclosing struct.

\*/

def lensed[B](lens: Lens[B, A]): ValueHydrator[B, C] =

ValueHydrator[B, C] { (b, ctx) =>

this.run(lens.get(b), ctx).map {

case ValueState(value, state) =>

ValueState(lens.set(b, value), state)

}

}

}

object ValueHydrator {

/\*\*

\* Create a ValueHydrator from a function that returns Stitch[ValueState[A]]

\*/

def apply[A, C](f: (A, C) => Stitch[ValueState[A]]): ValueHydrator[A, C] =

new ValueHydrator[A, C](f)

/\*\*

\* Produces a ValueState instance with the given value and an empty HydrationState

\*/

def unit[A, C]: ValueHydrator[A, C] =

ValueHydrator { (a, \_) => Stitch.value(ValueState.unit(a)) }

/\*\*

\* Runs several ValueHydrators in sequence.

\*/

def inSequence[A, C](bs: ValueHydrator[A, C]\*): ValueHydrator[A, C] =

bs match {

case Seq(b) => b

case Seq(b1, b2) => b1.andThen(b2)

case \_ => bs.reduceLeft(\_.andThen(\_))

}

/\*\*

\* Creates a `ValueHydrator` from a Mutation. If the mutation returns None (indicating

\* no change) the hydrator will return an ValueState.unmodified with the input value;

\* otherwise, it will return an ValueState.modified with the mutated value.

\* If the mutation throws an exception, it will be caught and lifted to Stitch.exception.

\*/

def fromMutation[A, C](mutation: Mutation[A]): ValueHydrator[A, C] =

ValueHydrator[A, C] { (input, \_) =>

Stitch.const(

Try {

mutation(input) match {

case None => ValueState.unmodified(input)

case Some(output) => ValueState.modified(output)

}

}

)

}

/\*\*

\* Creates a Hydrator from a non-`Stitch` producing function. If the function throws

\* an error it will be caught and converted to a Throw.

\*/

def map[A, C](f: (A, C) => ValueState[A]): ValueHydrator[A, C] =

ValueHydrator[A, C] { (a, ctx) => Stitch.const(Try(f(a, ctx))) }

}