/\*\*

\* Provides the ability to partially tee traffic to a secondary

\* service.

\*

\* This code was originally written to provide a way to provide

\* production traffic to the TweetyPie staging cluster, selecting a

\* consistent subset of tweet ids, to enable a production-like cache

\* hit rate with a much smaller cache.

\*/

package com.twitter.servo.forked

import com.twitter.servo.data.Lens

object Forked {

/\*\*

\* A strategy for executing forked actions.

\*/

type Executor = (() => Unit) => Unit

/\*\*

\* Directly execute the forked action.

\*/

val inlineExecutor: Executor = f => f()

/\*\*

\* Produce objects of type A to send to a secondary target.

\* Returning None signifies that nothing should be forked.

\*/

type Fork[A] = A => Option[A]

/\*\*

\* Fork the input unchanged, only when it passes the specified

\* predicate.

\*

\* For instance, if your service has a get() method

\*/

def forkWhen[T](f: T => Boolean): Fork[T] =

a => if (f(a)) Some(a) else None

/\*\*

\* Fork a subset of the elements of the Seq, based on the supplied

\* predicate. If the resulting Seq is empty, the secondary action

\* will not be executed.

\*/

def forkSeq[T](f: T => Boolean): Fork[Seq[T]] = { xs =>

val newXs = xs filter f

if (newXs.nonEmpty) Some(newXs) else None

}

/\*\*

\* Apply forking through lens.

\*/

def forkLens[A, B](lens: Lens[A, B], f: Fork[B]): Fork[A] =

a => f(lens(a)).map(lens.set(a, \_))

/\*\*

\* A factory for building actions that will partially tee their input

\* to a secondary target. The executor is parameterized to make the

\* execution strategy independent from the forking logic.

\*/

def toSecondary[S](secondary: S, executor: Executor): S => Forked[S] =

primary =>

new Forked[S] {

/\*\*

\* Tee a subset of requests defined by the forking function to the

\* secondary service.

\*/

def apply[Q, R](fork: Forked.Fork[Q], action: (S, Q) => R): Q => R = { req =>

fork(req) foreach { req =>

executor(() => action(secondary, req))

}

action(primary, req)

}

}

/\*\*

\* A forked action builder that bypasses the forking altogether and

\* just calls the supplied action on a service.

\*

\* This is useful for configurations that will sometimes have fork

\* targets defined and sometimes not.

\*/

def notForked[S]: S => Forked[S] =

service =>

new Forked[S] {

def apply[Q, R](unusedFork: Forked.Fork[Q], action: (S, Q) => R): Q => R =

action(service, \_)

}

}

/\*\*

\* Factory for forking functions, primarily useful for sending a copy

\* of a stream of requests to a secondary service.

\*/

trait Forked[S] {

import Forked.\_

/\*\*

\* Fork an action that takes two parameters, forking only on the

\* first parameter, passing the second unchanged.

\*/

def first[Q1, Q2, R](

fork: Fork[Q1],

action: S => (Q1, Q2) => R

): (Q1, Q2) => R = {

val f =

apply[(Q1, Q2), R](

fork = p =>

fork(p.\_1) map { q1 =>

(q1, p.\_2)

},

action = (svc, p) => action(svc)(p.\_1, p.\_2)

)

(q1, q2) => f((q1, q2))

}

def apply[Q, R](fork: Fork[Q], action: (S, Q) => R): Q => R

}