package com.twitter.servo.repository

import com.twitter.servo.util.RetryHandler

import com.twitter.util.{Duration, Future, Timer}

object Repository {

/\*\*

\* Composes a RepositoryFilter onto a Repository, producing a new Repository.

\*/

def composed[Q, R1, R2](

repo: Repository[Q, R1],

filter: RepositoryFilter[Q, R1, R2]

): Repository[Q, R2] =

q => filter(q, repo(q))

/\*\*

\* Chains 2 or more RepositoryFilters together into a single RepositoryFilter.

\*/

def chained[Q, R1, R2, R3](

f1: RepositoryFilter[Q, R1, R2],

f2: RepositoryFilter[Q, R2, R3],

fs: RepositoryFilter[Q, R3, R3]\*

): RepositoryFilter[Q, R1, R3] = {

val first: RepositoryFilter[Q, R1, R3] = (q, r) => f2(q, f1(q, r))

fs.toList match {

case Nil => first

case head :: tail => chained(first, head, tail: \_\*)

}

}

/\*\*

\* Wraps a Repository with a function that transforms queries on the way in, and

\* results on the way out.

\*/

def transformed[Q, Q2, R, R2](

repo: Repository[Q, R],

qmapper: Q2 => Q = (identity[Q] \_): (Q => Q),

rmapper: R => R2 = (identity[R] \_): (R => R)

): Repository[Q2, R2] =

qmapper andThen repo andThen { \_ map rmapper }

/\*\*

\* Wraps a Repository with another Repository that explodes the query into multiple

\* queries, executes those queries in parallel, then combines (reduces) results.

\*/

def mapReduced[Q, Q2, R, R2](

repo: Repository[Q, R],

mapper: Q2 => Seq[Q],

reducer: Seq[R] => R2

): Repository[Q2, R2] =

mapReducedWithQuery(repo, mapper, (rs: Seq[(Q, R)]) => reducer(rs map { case (\_, r) => r }))

/\*\*

\* An extension of mapReduced that passes query and result to the reducer.

\*/

def mapReducedWithQuery[Q, Q2, R, R2](

repo: Repository[Q, R],

mapper: Q2 => Seq[Q],

reducer: Seq[(Q, R)] => R2

): Repository[Q2, R2] = {

val queryRepo: Q => Future[(Q, R)] = q => repo(q) map { (q, \_) }

q2 => Future.collect(mapper(q2) map queryRepo) map reducer

}

/\*\*

\* Creates a new Repository that dispatches to r1 if the given query predicate returns true,

\* and dispatches to r2 otherwise.

\*/

def selected[Q, R](

select: Q => Boolean,

onTrueRepo: Repository[Q, R],

onFalseRepo: Repository[Q, R]

): Repository[Q, R] =

dispatched(select andThen {

case true => onTrueRepo

case false => onFalseRepo

})

/\*\*

\* Creates a new Repository that uses a function that selects an underlying repository

\* based upon the query.

\*/

def dispatched[Q, R](f: Q => Repository[Q, R]): Repository[Q, R] =

q => f(q)(q)

/\*\*

\* Wraps a Repository with the given RetryHandler, which may automatically retry

\* failed requests.

\*/

def retrying[Q, R](handler: RetryHandler[R], repo: Repository[Q, R]): Repository[Q, R] =

handler.wrap(repo)

/\*\*

\* Produces a new Repository where the returned Future must complete within the specified

\* timeout, otherwise the Future fails with a com.twitter.util.TimeoutException.

\*

\* ''Note'': On timeout, the underlying future is not interrupted.

\*/

def withTimeout[Q, R](

timer: Timer,

timeout: Duration,

repo: Repository[Q, R]

): Repository[Q, R] =

repo andThen { \_.within(timer, timeout) }

/\*\*

\* Produces a new Repository where the returned Future must complete within the specified

\* timeout, otherwise the Future fails with the specified Throwable.

\*

\* ''Note'': On timeout, the underlying future is not interrupted.

\*/

def withTimeout[Q, R](

timer: Timer,

timeout: Duration,

exc: => Throwable,

repo: Repository[Q, R]

): Repository[Q, R] =

repo andThen { \_.within(timer, timeout, exc) }

/\*\*

\* Wraps a Repository with stats recording functionality.

\*/

def observed[Q, R](

repo: Repository[Q, R],

observer: RepositoryObserver

): Repository[Q, R] =

query => {

observer.time() {

repo(query).respond(observer.observeTry)

}

}

}