package com.twitter.servo.request

import com.twitter.finagle.tracing.TraceId

import com.twitter.servo.util.{FunctionArrow, Effect, FutureArrow, FutureEffect, Observable}

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

/\*\*

\* Useful mixins for request types.

\*/

trait HasTraceId {

/\*\*

\* The Finagle TraceId of the request.

\*/

def traceId: TraceId

}

/\*\*

\* A collection of RequestFilter factory functions.

\*

\* type RequestFilter[A] = FutureArrow[A, A]

\*/

object RequestFilter {

/\*\*

\* Produce a RequestFilter from a function `A => Future[A]`.

\*/

def apply[A](f: A => Future[A]): RequestFilter[A] = FutureArrow(f)

/\*\*

\* Produce a RequestFilter from a function `A => Try[A]`.

\*

\* The Try is evaluated within a Future. Thus, Throw results are translated

\* to `Future.exception`s.

\*/

def fromTry[A](f: A => Try[A]): RequestFilter[A] = FutureArrow.fromTry(f)

/\*\*

\* A no-op RequestFilter; it simply returns the request.

\*

\* This forms a monoid with `append`.

\*/

def identity[A]: RequestFilter[A] = FutureArrow.identity

/\*\*

\* Appends two RequestFilters together.

\*

\* This forms a monoid with 'identity'.

\*/

def append[A](a: RequestFilter[A], b: RequestFilter[A]): RequestFilter[A] =

FutureArrow.append(a, b)

/\*\*

\* Compose an ordered series of RequestFilters into a single object.

\*/

def all[A](filters: RequestFilter[A]\*): RequestFilter[A] =

filters.foldLeft(identity[A])(append)

/\*\*

\* Produce a RequestFilter that applies a side-effect, returning the argument

\* request as-is.

\*/

def effect[A](effect: Effect[A]): RequestFilter[A] =

FutureArrow.fromFunctionArrow(FunctionArrow.effect(effect))

/\*\*

\* Produce a RequestFilter that applies a side-effect, returning the argument

\* request as-is.

\*/

def effect[A](effect: FutureEffect[A]): RequestFilter[A] = FutureArrow.effect(effect)

/\*\*

\* Returns a new request filter where all Futures returned from `a` have their

\* `masked` method called

\*/

def masked[A](a: RequestFilter[A]): RequestFilter[A] = a.masked

/\*\*

\* Produces a RequestFilter that proxies to one of two others, depending on a

\* predicate.

\*/

def choose[A](

predicate: A => Boolean,

ifTrue: RequestFilter[A],

ifFalse: RequestFilter[A]

): RequestFilter[A] =

FutureArrow.choose(predicate, ifTrue, ifFalse)

/\*\*

\* Guard the application of a filter on a predicate. The filter is applied

\* if the predicate returns true, otherwise, the request is simply returned.

\*/

def onlyIf[A](predicate: A => Boolean, f: RequestFilter[A]): RequestFilter[A] =

FutureArrow.onlyIf(predicate, f)

/\*\*

\* Produces a RequestFilter that authorizes requests by applying an

\* authorization function `A => Future[Unit]`. If the authorizer function

\* results in a Future exception, requests are failed. Otherwise, they pass.

\*/

def authorized[A <: Observable](authorizer: ClientRequestAuthorizer): RequestFilter[A] =

RequestFilter[A] { request =>

authorizer(request.requestName, request.clientIdString) map { \_ =>

request

}

}

/\*\*

\* Produces a RequestFilter that applies a ClientRequestObserver to requests.

\*

\* Used to increment counters and track stats for requests.

\*/

def observed[A <: Observable](observer: ClientRequestObserver): RequestFilter[A] =

RequestFilter[A] { request =>

val clientIdScopesOpt = request.clientIdString map { Seq(\_) }

observer(request.requestName, clientIdScopesOpt) map { \_ =>

request

}

}

}