package com.twitter.servo.request

import com.twitter.servo.gate.RateLimitingGate

import com.twitter.servo.util.Gate

import com.twitter.util.Future

/\*\*

\* Collects per-request stats by method-name and client.

\*/

trait ClientRequestAuthorizer extends ((String, Option[String]) => Future[Unit]) { self =>

/\*\*

\* @param methodName the name of the Service method being called

\* @param clientIdStrOpt an Option of the string value of the originating

\* request's ClientId

\*/

def apply(methodName: String, clientIdStrOpt: Option[String]): Future[Unit]

/\*\*

\* Compose this authorizer with another so that one is applied after the other.

\*

\* The resultant authorizer requires both underlying authorizers to succeed in

\* order to authorize a request.

\*/

def andThen(other: ClientRequestAuthorizer) = new ClientRequestAuthorizer {

override def apply(methodName: String, clientIdStrOpt: Option[String]): Future[Unit] = {

self.apply(methodName, clientIdStrOpt) flatMap { \_ =>

other(methodName, clientIdStrOpt)

}

}

}

}

object ClientRequestAuthorizer {

case class UnauthorizedException(msg: String) extends Exception(msg)

protected[this] val noClientIdException =

Future.exception(new UnauthorizedException("No ClientId specified"))

protected[this] val unauthorizedException =

new UnauthorizedException("Your ClientId is not authorized.")

protected[this] val overRateLimitException =

new UnauthorizedException("Your ClientId is over the allowed rate limit.")

/\*\*

\* Increment stats counters for this request.

\*

\* Note that ClientRequestAuthorizer.observed doesn't compose in the same fashion

\* as other authorizers via `andThen`. In order to observe authorization results,

\* pass in an underlying authorizer as an argument to observed.

\*/

def observed(

underlyingAuthorizer: ClientRequestAuthorizer,

observer: ClientRequestObserver

) = new ClientRequestAuthorizer {

override def apply(methodName: String, clientIdStrOpt: Option[String]): Future[Unit] = {

val clientIdStr = clientIdStrOpt.getOrElse("no\_client\_id")

observer(methodName, clientIdStrOpt map { Seq(\_) })

underlyingAuthorizer(methodName, clientIdStrOpt) onFailure { \_ =>

observer.unauthorized(methodName, clientIdStr)

} onSuccess { \_ =>

observer.authorized(methodName, clientIdStr)

}

}

}

def observed(observer: ClientRequestObserver): ClientRequestAuthorizer =

observed(ClientRequestAuthorizer.permissive, observer)

/\*\*

\* Lets all requests through.

\*/

def permissive = new ClientRequestAuthorizer {

override def apply(methodName: String, clientIdStrOpt: Option[String]) = Future.Done

}

/\*\*

\* A Generic Authorizer that allows you to pass in your own authorizer function (filter).

\* The filter should take in methodName and clientId and return a Boolean decision

\*

\* Note: Requires requests to have ClientIds.

\* @param exception return this exception if the request does not pass the filter

\*/

def filtered(

filter: (String, String) => Boolean,

exception: Exception = unauthorizedException

): ClientRequestAuthorizer =

new ClientRequestAuthorizer {

val futureException = Future.exception(exception)

override def apply(methodName: String, clientIdStrOpt: Option[String]): Future[Unit] = {

clientIdStrOpt match {

case Some(clientIdStr) =>

if (filter(methodName, clientIdStr))

Future.Done

else

futureException

case None =>

noClientIdException

}

}

}

/\*\*

\* Authorizes client requests based on a allowlist of ClientId strings.

\*/

def allowlisted(allowlist: Set[String]): ClientRequestAuthorizer =

filtered { (\_, clientIdStr) =>

allowlist.contains(clientIdStr)

}

/\*\*

\* Authorizes requests if and only if they have an associated ClientId.

\*/

def withClientId: ClientRequestAuthorizer = filtered { (\_, \_) =>

true

}

/\*\*

\* Consult a (presumably) Decider-backed predicate to authorize requests by ClientId.

\* @param exception return this exception if the request does not pass the filter

\*/

def deciderable(

isAvailable: String => Boolean,

exception: Exception = unauthorizedException

): ClientRequestAuthorizer =

filtered(

{ (\_, clientIdStr) =>

isAvailable(clientIdStr)

},

exception

)

/\*\*

\* Simple rate limiter for unknown client ids. Useful for letting new clients

\* send some traffic without the risk of being overrun by requests.

\*

\* @param limitPerSecond Number of calls per second we can tolerate

\*/

def rateLimited(limitPerSecond: Double): ClientRequestAuthorizer = {

gated(RateLimitingGate.uniform(limitPerSecond), overRateLimitException)

}

/\*\*

\* Simple Gate based authorizer, will authorize according to the result of the gate regardless

\* of the client/method name

\*/

def gated(

gate: Gate[Unit],

exception: Exception = unauthorizedException

): ClientRequestAuthorizer = {

deciderable(\_ => gate(), exception)

}

/\*\*

\* @return A ClientRequestAuthorizer that switches between two provided

\* ClientRequestAuthorizers depending on a decider.

\*/

def select(

decider: Gate[Unit],

ifTrue: ClientRequestAuthorizer,

ifFalse: ClientRequestAuthorizer

): ClientRequestAuthorizer =

new ClientRequestAuthorizer {

override def apply(methodName: String, clientIdStrOpt: Option[String]): Future[Unit] =

decider.pick(

ifTrue(methodName, clientIdStrOpt),

ifFalse(methodName, clientIdStrOpt)

)

}

}