package com.twitter.tweetypie

package service

/\*\*

\* An authorizer for determining if a request to a

\* method should be rejected.

\*

\* This class is in the spirit of servo.request.ClientRequestAuthorizer.

\* The difference is ClientRequestAuthorizer only operates

\* on two pieces of information, clientId and a method name.

\*

\* This class can be used to create a more complex authorizer that

\* operates on the specifics of a request. e.g, an

\* authorizer that disallows certain clients from passing

\* certain optional flags.

\*

\* Note: With some work, ClientRequestAuthorizer could be

\* generalized to support cases like this. If we end up making

\* more method authorizers it might be worth it to

\* go that route.

\*/

abstract class MethodAuthorizer[T]() {

def apply(request: T, clientId: String): Future[Unit]

/\*\*

\* Created decidered MethodAuthorizer

\* if the decider is off it will execute

\* MethodAuthorizer.unit, which always succeeds.

\*/

def enabledBy(decider: Gate[Unit]): MethodAuthorizer[T] =

MethodAuthorizer.select(decider, this, MethodAuthorizer.unit)

/\*\*

\* Transform this MethodAuthorizer[T] into a MethodAuthorizer[A]

\* by providing a function from A => T

\*/

def contramap[A](f: A => T): MethodAuthorizer[A] =

MethodAuthorizer[A] { (request, clientId) => this(f(request), clientId) }

}

object MethodAuthorizer {

/\*\*

\* @param f an authorization function that returns

\* Future.Unit if the request is authorized, and Future.exception()

\* if the request is not authorized.

\*

\* @return An instance of MethodAuthorizer with an apply method

\* that returns f

\*/

def apply[T](f: (T, String) => Future[Unit]): MethodAuthorizer[T] =

new MethodAuthorizer[T]() {

def apply(request: T, clientId: String): Future[Unit] = f(request, clientId)

}

/\*\*

\* @param authorizers A seq of MethodAuthorizers to be

\* composed into one.

\* @return A MethodAuthorizer that sequentially executes

\* all of the authorizers

\*/

def all[T](authorizers: Seq[MethodAuthorizer[T]]): MethodAuthorizer[T] =

MethodAuthorizer { (request, clientId) =>

authorizers.foldLeft(Future.Unit) {

case (f, authorize) => f.before(authorize(request, clientId))

}

}

/\*\*

\* @return A MethodAuthorizer that always returns Future.Unit

\* Useful if you need to decider off your MethodAuthorizer

\* and replace it with one that always passes.

\*/

def unit[T]: MethodAuthorizer[T] = MethodAuthorizer { (request, client) => Future.Unit }

/\*\*

\* @return A MethodAuthorizer that switches between two provided

\* MethodAuthorizers depending on a decider.

\*/

def select[T](

decider: Gate[Unit],

ifTrue: MethodAuthorizer[T],

ifFalse: MethodAuthorizer[T]

): MethodAuthorizer[T] =

MethodAuthorizer { (request, client) =>

decider.pick(

ifTrue(request, client),

ifFalse(request, client)

)

}

}