package com.twitter.servo.request

import com.twitter.config.yaml.YamlMap

import com.twitter.util.Try

/\*\*

\* Module for defining a set of permissions. This is similar to

\* Enumeration in the scala standard library.

\*

\* To use, instantiate a subclass:

\*

\* {{{

\* object MyPermissions extends PermissionModule {

\* val Eat = create("eat")

\* val Drink = create("drink")

\* }

\* }}}

\*

\* Permissions only support one kind of authorization, which is that

\* you can check whether a holder of permissions has all of the

\* permissions in a particular set.

\*

\* {{{

\* val snack = MyPermissions.Eat

\* val dinner = MyPermissions.Eat union MyPermissions.Drink

\* val canEat = MyPermissions.Eat

\* dinner satisfiedBy canEat // false

\* snack satisfiedBy canEat // true

\* }}}

\*

\* Each instance will have its own distinct permission type, so it is

\* not possible to confuse the permissions defined in different

\* modules.

\*

\* {{{

\* scala> object P1 extends PermissionModule { val Read = create("read") }

\* scala> object P2 extends PermissionModule { val Read = create("read") }

\* scala> P1.Read satisfiedBy P2.Read

\* error: type mismatch;

\* found : P2.Permissions

\* required: P1.Permissions

\* P1.Read satisfiedBy P2.Read

\* }}}

\*

\* Once an instance has been created, it will not be possible to

\* create new permissions. The intention is that all permissions will

\* be created at object initialization time.

\*

\* Each instance also supplies functionality for accessing permissions

\* by name, including parsing client permission maps from YAML.

\*/

trait PermissionModule {

// This var is used during object initialization to collect all of

// the permissions that are created in the subclass. The lazy

// initializer for `All` will set this to null as a side-effect, so

// that further permission creations are not allowed.

@volatile private[this] var allPerms: Set[String] = Set.empty

/\*\*

\* Create a new Permission with the given name. Note that "\*" is a

\* reversed string for `All` permissions, thus it can not be

\* used as the name of an individual permission.

\*

\* This method must be called before `All` is accessed.

\* The intention is that it should be called as part of

\* object initialization.

\*

\* Note that some methods of PermissionModule access `All`, so it is

\* best to create all of your permissions before doing anything

\* else.

\*

\* @throws RuntimeException: If it is called after `All` has been

\* initialized.

\*/

protected def create(name: String) = {

synchronized {

if (allPerms == null) {

throw new RuntimeException("Permission creation after initialization")

}

allPerms = allPerms union Set(name)

}

new Permissions(Set(name))

}

/\*\*

\* Get a set of permissions with this single permission by name. It

\* will return None if there is no permission by that name.

\*

\* No permissions may be defined after this method is called.

\*/

def get(name: String): Option[Permissions] = All.get(name)

/\*\*

\* Get the set of permissions that contains that single permission

\* by name.

\*

\* @throws RuntimeException if there is no defined permission with

\* this name.

\*

\* No permissions may be defined after this method is called.

\*/

def apply(name: String): Permissions =

get(name) match {

case None => throw new RuntimeException("Unknown permission: " + name)

case Some(p) => p

}

/\*\*

\* No permissions (required or held)

\*/

val Empty: Permissions = new Permissions(Set.empty)

/\*\*

\* All defined permissions.

\*

\* No permissions may be defined after this value is initialized.

\*/

lazy val All: Permissions = {

val p = new Permissions(allPerms)

allPerms = null

p

}

/\*\*

\* Load permissions from a YAML map.

\*

\* No permissions may be defined after this method is called.

\*

\* @return a map from client identifier to permission set.

\* @throws RuntimeException when the permission from the Map is not defined.

\*/

def fromYaml(m: YamlMap): Try[Map[String, Permissions]] =

Try {

m.keys.map { k =>

k -> fromSeq((m yamlList k).map { \_.toString })

}.toMap

}

/\*\*

\* Load permissions from map.

\*

\* No permissions may be defined after this method is called.

\*

\* @param m a map from client identifier to a set of permission strings

\*

\* @return a map from client identifier to permission set.

\* @throws RuntimeException when the permission from the Map is not defined.

\*/

def fromMap(m: Map[String, Seq[String]]): Try[Map[String, Permissions]] =

Try {

m.map { case (k, v) => k -> fromSeq(v) }

}

/\*\*

\* Load permissions from seq.

\*

\* No permissions may be defined after this method is called.

\*

\* @param sequence a Seq of permission strings

\*

\* @return a permission set.

\* @throws RuntimeException when the permission is not defined.

\*/

def fromSeq(permissionStrings: Seq[String]): Permissions =

permissionStrings.foldLeft(Empty) { (p, v) =>

v match {

case "all" if get("all").isEmpty => All

case other => p union apply(other)

}

}

/\*\*

\* Authorizer based on a Permissions for RPC method names.

\* @param requiredPermissions

\* map of RPC method names to Permissions required for that RPC

\* @param clientPermissions

\* map of ClientId to Permissions a client has

\*/

def permissionBasedAuthorizer(

requiredPermissions: Map[String, Permissions],

clientPermissions: Map[String, Permissions]

): ClientRequestAuthorizer =

ClientRequestAuthorizer.filtered { (methodName, clientId) =>

requiredPermissions.get(methodName) exists {

\_ satisfiedBy clientPermissions.getOrElse(clientId, Empty)

}

}

/\*\*

\* A set of permissions. This can represent either permissions that

\* are required to perform an action, or permissions that are held

\* by a client.

\*

\* This type cannot be instantiated directly. Use the methods of

\* your subclass of PermissionModule to do so.

\*/

class Permissions private[PermissionModule] (private[PermissionModule] val permSet: Set[String]) {

/\*\*

\* Does the supplied set of held permissions satisfy the

\* requirements of this set of permissions?

\*

\* For example, if this set of permissions is Set("read"), and the

\* other set of permissions is Set("read", "write"), then the

\* other set of permissions satisfies this set.

\*/

def satisfiedBy(other: Permissions): Boolean = permSet subsetOf other.permSet

override def equals(other: Any): Boolean =

other match {

case p: Permissions => p.permSet == permSet

case \_ => false

}

override lazy val hashCode: Int = 5 + 37 \* permSet.hashCode

/\*\*

\* Get a single permission

\*/

def get(permName: String): Option[Permissions] =

if (permSet contains permName) Some(new Permissions(Set(permName))) else None

/\*\*

\* Create a new permission set that holds the permissions of this

\* object as well as the permissions of the other object.

\*/

def union(other: Permissions): Permissions = new Permissions(permSet union other.permSet)

override def toString: String = "Permissions(%s)".format(permSet.mkString(", "))

}

}