package com.twitter.tweetypie

package config

import com.twitter.finagle.mtls.authentication.ServiceIdentifier

import com.twitter.finagle.mtls.transport.S2STransport

import com.twitter.servo.gate.RateLimitingGate

import com.twitter.servo.request.ClientRequestAuthorizer.UnauthorizedException

import com.twitter.servo.request.{ClientRequestAuthorizer, ClientRequestObserver}

import com.twitter.tweetypie.client\_id.ClientIdHelper

import com.twitter.tweetypie.client\_id.PreferForwardedServiceIdentifierForStrato

import com.twitter.tweetypie.core.RateLimited

import com.twitter.tweetypie.service.MethodAuthorizer

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.util.Future

/\*\*

\* Compose a ClientRequestAuthorizer for

\* ClientHandlingTweetService

\*/

object ClientHandlingTweetServiceAuthorizer {

private val RateLimitExceeded =

RateLimited("Your ClientId has exceeded the rate limit for non-allowListed clients.")

def apply(

settings: TweetServiceSettings,

dynamicConfig: DynamicConfig,

statsReceiver: StatsReceiver,

getServiceIdentifier: () => ServiceIdentifier = S2STransport.peerServiceIdentifier \_

): ClientRequestAuthorizer = {

val authorizer =

if (settings.allowlistingRequired) {

val limitingGate = RateLimitingGate.uniform(settings.nonAllowListedClientRateLimitPerSec)

allowListedOrRateLimitedAuthorizer(dynamicConfig, limitingGate)

.andThen(rejectNonAllowListedProdAuthorizer(dynamicConfig))

.andThen(permittedMethodsAuthorizer(dynamicConfig))

.andThen(allowProductionAuthorizer(settings.allowProductionClients))

} else {

ClientRequestAuthorizer.withClientId

}

val alternativeClientIdHelper = new ClientIdHelper(PreferForwardedServiceIdentifierForStrato)

// pass the authorizer into an observed authorizer for stats tracking.

// (observed authorizers can't be composed with andThen)

ClientRequestAuthorizer.observed(

authorizer,

new ClientRequestObserver(statsReceiver) {

override def apply(

methodName: String,

clientIdScopesOpt: Option[Seq[String]]

): Future[Unit] = {

// Monitor for the migration taking into account forwarded service identifier

// as effective client ID for strato.

val alternativeClientIdScopes = alternativeClientIdHelper.effectiveClientId.map(Seq(\_))

if (clientIdScopesOpt != alternativeClientIdScopes) {

scopedReceiver.scope(methodName)

.scope("before\_migration")

.scope(clientIdScopesOpt.getOrElse(Seq(ClientIdHelper.UnknownClientId)): \_\*)

.scope("after\_migration")

.counter(alternativeClientIdScopes.getOrElse(Seq(ClientIdHelper.UnknownClientId)): \_\*)

.incr()

} else {

scopedReceiver.scope(methodName).counter("migration\_indifferent").incr()

}

super.apply(methodName, clientIdScopesOpt)

}

override def authorized(methodName: String, clientIdStr: String): Unit = {

// Monitor for the migration of using service identifier

// as identity instead of client ID.

val serviceIdentifier = getServiceIdentifier()

scopedReceiver.counter(

"authorized\_request",

clientIdStr,

serviceIdentifier.role,

serviceIdentifier.service,

serviceIdentifier.environment

).incr()

val status = dynamicConfig.byServiceIdentifier(serviceIdentifier).toSeq match {

case Seq() => "none"

case Seq(client) if client.clientId == clientIdStr => "equal"

case Seq(\_) => "other"

case \_ => "ambiguous"

}

scopedReceiver.counter(

"service\_id\_match\_client\_id",

clientIdStr,

serviceIdentifier.role,

serviceIdentifier.service,

serviceIdentifier.environment,

status

).incr()

}

}

)

}

/\*\*

\* @return A ClientRequestAuthorizer that allows unlimited requests for allowlisted client ids and

\* rate-limited requests for unknown clients.

\*/

def allowListedOrRateLimitedAuthorizer(

dynamicConfig: DynamicConfig,

nonAllowListedLimiter: Gate[Unit]

): ClientRequestAuthorizer =

ClientRequestAuthorizer.filtered(

{ (\_, clientId) =>

dynamicConfig.isAllowListedClient(clientId) || nonAllowListedLimiter()

},

RateLimitExceeded)

/\*\*

\* @return A ClientRequestAuthorizer that rejects requests from non-allowListed prod clients.

\*/

def rejectNonAllowListedProdAuthorizer(dynamicConfig: DynamicConfig): ClientRequestAuthorizer = {

object UnallowlistedException

extends UnauthorizedException(

"Traffic is only allowed from allow-listed \*.prod clients." +

" Please create a ticket to register your clientId to enable production traffic using http://go/tp-new-client."

)

def isProdClient(clientId: String): Boolean =

clientId.endsWith(".prod") || clientId.endsWith(".production")

ClientRequestAuthorizer.filtered(

{ (\_, clientId) =>

!isProdClient(clientId) || dynamicConfig.isAllowListedClient(clientId)

},

UnallowlistedException)

}

/\*\*

\* @return A ClientRequestAuthorizer that checks if a given client's

\* permittedMethods field includes the method they are calling

\*/

def permittedMethodsAuthorizer(dynamicConfig: DynamicConfig): ClientRequestAuthorizer =

dynamicConfig.clientsByFullyQualifiedId match {

case Some(clientsById) => permittedMethodsAuthorizer(dynamicConfig, clientsById)

case None => ClientRequestAuthorizer.permissive

}

private def permittedMethodsAuthorizer(

dynamicConfig: DynamicConfig,

clientsByFullyQualifiedId: Map[String, Client]

): ClientRequestAuthorizer = {

ClientRequestAuthorizer.filtered { (methodName, clientId) =>

dynamicConfig.unprotectedEndpoints(methodName) ||

(clientsByFullyQualifiedId.get(clientId) match {

case Some(client) =>

client.accessAllMethods ||

client.permittedMethods.contains(methodName)

case None =>

false // If client id is unknown, don't allow access

})

}

}

/\*\*

\* @return A ClientRequestAuthorizer that fails the

\* request if it is coming from a production client

\* and allowProductionClients is false

\*/

def allowProductionAuthorizer(allowProductionClients: Boolean): ClientRequestAuthorizer =

ClientRequestAuthorizer.filtered { (\_, clientId) =>

allowProductionClients || !(clientId.endsWith(".prod") || clientId.endsWith(".production"))

}

}

/\*\*

\* Compose a MethodAuthorizer for the `getTweets` endpoint.

\*/

object GetTweetsAuthorizer {

import ProtectedTweetsAuthorizer.IncludeProtected

def apply(

config: DynamicConfig,

maxRequestSize: Int,

instanceCount: Int,

enforceRateLimitedClients: Gate[Unit],

maxRequestWidthEnabled: Gate[Unit],

statsReceiver: StatsReceiver,

): MethodAuthorizer[GetTweetsRequest] =

MethodAuthorizer.all(

Seq(

ProtectedTweetsAuthorizer(config.clientsByFullyQualifiedId)

.contramap[GetTweetsRequest] { r =>

IncludeProtected(r.options.exists(\_.bypassVisibilityFiltering))

},

RequestSizeAuthorizer(maxRequestSize, maxRequestWidthEnabled)

.contramap[GetTweetsRequest](\_.tweetIds.size),

RateLimiterAuthorizer(config, instanceCount, enforceRateLimitedClients, statsReceiver)

.contramap[GetTweetsRequest](\_.tweetIds.size)

)

)

}

/\*\*

\* Compose a MethodAuthorizer for the `getTweetFields` endpoint.

\*/

object GetTweetFieldsAuthorizer {

import ProtectedTweetsAuthorizer.IncludeProtected

def apply(

config: DynamicConfig,

maxRequestSize: Int,

instanceCount: Int,

enforceRateLimitedClients: Gate[Unit],

maxRequestWidthEnabled: Gate[Unit],

statsReceiver: StatsReceiver

): MethodAuthorizer[GetTweetFieldsRequest] =

MethodAuthorizer.all(

Seq(

ProtectedTweetsAuthorizer(config.clientsByFullyQualifiedId)

.contramap[GetTweetFieldsRequest](r =>

IncludeProtected(r.options.visibilityPolicy == TweetVisibilityPolicy.NoFiltering)),

RequestSizeAuthorizer(maxRequestSize, maxRequestWidthEnabled)

.contramap[GetTweetFieldsRequest](\_.tweetIds.size),

RateLimiterAuthorizer(config, instanceCount, enforceRateLimitedClients, statsReceiver)

.contramap[GetTweetFieldsRequest](\_.tweetIds.size)

)

)

}

object ProtectedTweetsAuthorizer {

case class IncludeProtected(include: Boolean) extends AnyVal

class BypassVisibilityFilteringNotAuthorizedException(message: String)

extends UnauthorizedException(message)

def apply(optClientsById: Option[Map[String, Client]]): MethodAuthorizer[IncludeProtected] = {

optClientsById match {

case Some(clientsByFullyQualifiedId) =>

val clientsWithBypassVisibilityFiltering = clientsByFullyQualifiedId.filter {

case (\_, client) => client.bypassVisibilityFiltering

}

apply(clientId => clientsWithBypassVisibilityFiltering.contains(clientId))

case None =>

apply((\_: String) => true)

}

}

/\*\*

\* A MethodAuthorizer that fails the request if a client requests to bypass visibility

\* filtering but doesn't have BypassVisibilityFiltering

\*/

def apply(protectedTweetsAllowlist: String => Boolean): MethodAuthorizer[IncludeProtected] =

MethodAuthorizer { (includeProtected, clientId) =>

// There is only one unauthorized case, a client requesting

// protected tweets when they are not in the allowlist

Future.when(includeProtected.include && !protectedTweetsAllowlist(clientId)) {

Future.exception(

new BypassVisibilityFilteringNotAuthorizedException(

s"$clientId is not authorized to bypass visibility filtering"

)

)

}

}

}

/\*\*

\* A MethodAuthorizer[Int] that fails large requests.

\*/

object RequestSizeAuthorizer {

class ExceededMaxWidthException(message: String) extends UnauthorizedException(message)

def apply(

maxRequestSize: Int,

maxWidthLimitEnabled: Gate[Unit] = Gate.False

): MethodAuthorizer[Int] =

MethodAuthorizer { (requestSize, clientId) =>

Future.when(requestSize > maxRequestSize && maxWidthLimitEnabled()) {

Future.exception(

new ExceededMaxWidthException(

s"$requestSize exceeds bulk request size limit. $clientId can request at most $maxRequestSize items per request"

)

)

}

}

}

object RateLimiterAuthorizer {

type ClientId = String

/\*\*

\* @return client ID to weighted RateLimitingGate map

\*

\* We want to rate-limit based on requests per sec for every instance.

\* When we allowlist new clients to Tweetypie, we assign tweets per sec quota.

\* That's why, we compute perInstanceQuota [1] and create a weighted rate-limiting gate [2]

\* which returns true if acquiring requestSize number of permits is successful. [3]

\*

\* [1] tps quota during allowlisting is for both DCs and instanceCount is for one DC.

\* Therefore, we are over-compensating perInstanceQuota for all low-priority clients.

\* this will act a fudge-factor to account for cluster-wide traffic imbalances.

\*

\* val perInstanceQuota : Double = math.max(1.0, math.ceil(tpsLimit.toFloat / instanceCount))

\*

\* We have some clients like deferredRPC with 0K tps quota and rate limiter expects > 0 permits.

\*

\* [2] if a client has multiple environments - staging, devel, prod. We provision the

\* same rate-limits for all envs instead of distributing the tps quota across envs.

\*

\* Example:

\*

\* val c = Client(..., limit = 10k, ...)

\* Map("foo.prod" -> c, "foo.staging" -> c, "foo.devel" -> c)

\*

\* Above client config turns into 3 separate RateLimitingGate.weighted(), each with 10k

\*

\* [3] RateLimitingGate will always give permit to the initial request that exceeds

\* the limit. ex: starting with rate-limit of 1 tps per instance. first request with

\* 100 batch size is allowed.

\*

\* RateLimitFudgeFactor is a multiplier for per-instance quota to account for:

\*

\* a) High likelihood of concurrent batches hitting the same tweetypie shard due to

\* non-uniform load distribution (this can be alleviated by using Deterministic Aperture)

\* b) Clients with no retry backoffs and custom batching/concurrency.

\*

\* We are adding default stitch batch size to per instance quota, to give more headroom for low-tps clients.

\* https://cgit.twitter.biz/source/tree/stitch/stitch-tweetypie/src/main/scala/com/twitter/stitch/tweetypie/TweetyPie.scala#n47

\*

\*/

case class RateLimiterConfig(limitingGate: Gate[Int], enforceRateLimit: Boolean)

def perClientRateLimiters(

dynamicConfig: DynamicConfig,

instanceCount: Int

): Map[ClientId, RateLimiterConfig] = {

val RateLimitFudgeFactor: Double = 1.5

val DefaultStitchBatchSize: Double = 25.0

dynamicConfig.clientsByFullyQualifiedId match {

case Some(clients) =>

clients.collect {

case (clientId, client) if client.tpsLimit.isDefined =>

val perInstanceQuota: Double =

math.max(

1.0,

math.ceil(

client.tpsLimit.get.toFloat / instanceCount)) \* RateLimitFudgeFactor + DefaultStitchBatchSize

clientId -> RateLimiterConfig(

RateLimitingGate.weighted(perInstanceQuota),

client.enforceRateLimit

)

}

case None => Map.empty

}

}

/\*

enforce rate-limiting on get\_tweets and get\_tweet\_fields requests

given enable\_rate\_limited\_clients decider is true and rate limiting gate

is not giving any more permits.

\*/

def apply(

config: DynamicConfig,

limiters: Map[ClientId, RateLimiterConfig],

instanceCount: Int,

enforceRateLimitedClients: Gate[Unit],

statsReceiver: StatsReceiver

): MethodAuthorizer[Int] = {

val tpsExceededScope = statsReceiver.scope("tps\_exceeded")

val tpsRejectedScope = statsReceiver.scope("tps\_rejected")

val qpsExceededScope = statsReceiver.scope("qps\_exceeded")

val qpsRejectedScope = statsReceiver.scope("qps\_rejected")

MethodAuthorizer { (requestSize, clientId) =>

val positiveRequestSize = math.max(1, requestSize)

val shouldRateLimit: Boolean = limiters.get(clientId).exists { config =>

val exceededLimit = !config.limitingGate(positiveRequestSize)

if (exceededLimit) {

qpsExceededScope.counter(clientId).incr()

tpsExceededScope.counter(clientId).incr(positiveRequestSize)

}

exceededLimit && config.enforceRateLimit

}

Future.when(shouldRateLimit && enforceRateLimitedClients()) {

qpsRejectedScope.counter(clientId).incr()

tpsRejectedScope.counter(clientId).incr(positiveRequestSize)

Future.exception(

RateLimited(s"Your client ID $clientId has exceeded its reserved tps quota.")

)

}

}

}

def apply(

config: DynamicConfig,

instanceCount: Int,

enforceRateLimitedClients: Gate[Unit],

statsReceiver: StatsReceiver

): MethodAuthorizer[Int] = {

val limiters = perClientRateLimiters(config, instanceCount)

apply(config, limiters, instanceCount, enforceRateLimitedClients, statsReceiver)

}

}