package com.twitter.tweetypie.config

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

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.tweetypie.Gate

import com.twitter.tweetypie.backends.ConfigBus

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

import com.twitter.util.Activity

case class DynamicConfig(

// A map of fully-qualified client ID (including the environment suffix, e.g. tweetypie.prod) to Client case class

clientsByFullyQualifiedId: Option[Map[String, Client]],

// Clients by service identifier parts.

clientsByRole: Option[Map[String, Seq[Client]]] = None,

clientsByService: Option[Map[String, Seq[Client]]] = None,

onlyEnvClients: Option[Seq[Client]] = None,

// These endpoints do not need permissions to be accessed

unprotectedEndpoints: Set[String] = Set("get\_tweet\_counts", "get\_tweet\_fields", "get\_tweets")) {

/\*\*

\* Function that takes a fully qualified client id and says whether it is included in the allowList

\*/

val isAllowListedClient: String => Boolean =

clientsByFullyQualifiedId.map(clients => clients.contains \_).getOrElse(\_ => true)

def byServiceIdentifier(serviceIdentifier: ServiceIdentifier): Set[Client] =

Iterable.concat(

get(clientsByRole, serviceIdentifier.role),

get(clientsByService, serviceIdentifier.service),

onlyEnvClients.getOrElse(Seq()),

)

.filter(\_.matches(serviceIdentifier))

.toSet

private def get(clientsByKey: Option[Map[String, Seq[Client]]], key: String): Seq[Client] =

clientsByKey match {

case Some(map) => map.getOrElse(key, Seq())

case None => Seq()

}

/\*\*

\* Take a fully qualified client id and says if the client has offered to shed reads if tweetypie

\* is in an emergency

\*/

val loadShedEligible: Gate[String] = Gate { (clientId: String) =>

val env = ClientIdHelper.getClientIdEnv(clientId)

clientsByFullyQualifiedId.flatMap(clients => clients.get(clientId)).exists { c =>

c.loadShedEnvs.contains(env)

}

}

}

/\*\*

\* DynamicConfig uses ConfigBus to update Tweetypie with configuration changes

\* dynamically. Every time the config changes, the Activity[DynamicConfig] is

\* updated, and anything relying on that config will be reinitialized.

\*/

object DynamicConfig {

def fullyQualifiedClientIds(client: Client): Seq[String] = {

val clientId = client.clientId

client.environments match {

case Nil => Seq(clientId)

case envs => envs.map(env => s"$clientId.$env")

}

}

// Make a Map of fully qualified client id to Client

def byClientId(clients: Seq[Client]): Map[String, Client] =

clients.flatMap { client =>

fullyQualifiedClientIds(client).map { fullClientId => fullClientId -> client }

}.toMap

def by(get: ServiceIdentifierPattern => Option[String])(clients: Seq[Client]): Map[String, Seq[Client]] =

clients.flatMap { c =>

c.serviceIdentifiers.collect {

case s if get(s).isDefined => (get(s).get, c)

}

}.groupBy(\_.\_1).mapValues(\_.map(\_.\_2))

private[this] val clientsPath = "config/clients.yml"

def apply(

stats: StatsReceiver,

configBus: ConfigBus,

settings: TweetServiceSettings

): Activity[DynamicConfig] =

DynamicConfigLoader(configBus.file)

.apply(clientsPath, stats.scope("client\_allowlist"), ClientsParser.apply)

.map(fromClients)

def fromClients(clients: Option[Seq[Client]]): DynamicConfig =

DynamicConfig(

clientsByFullyQualifiedId = clients.map(byClientId),

clientsByRole = clients.map(by(\_.role)),

clientsByService = clients.map(by(\_.service)),

onlyEnvClients = clients.map(\_.filter { client =>

client.serviceIdentifiers.exists(\_.onlyEnv)

}),

)

}