package com.twitter.tweetypie.context

import com.twitter.context.TwitterContext

import com.twitter.finagle.Filter

import com.twitter.finagle.Service

import com.twitter.finagle.SimpleFilter

import com.twitter.finagle.context.Contexts

import com.twitter.io.Buf

import com.twitter.io.Buf.ByteArray.Owned

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.graphql.common.core.GraphQlClientApplication

import com.twitter.util.Try

import java.nio.charset.StandardCharsets.UTF\_8

import scala.util.matching.Regex

/\*\*

\* Context and filters to help track callers of Tweetypie's endpoints. This context and its

\* filters were originally added to provide visibility into callers of Tweetypie who are

\* using the birdherd library to access tweets.

\*

\* This context data is intended to be marshalled by callers to Tweetypie, but then the

\* context data is stripped (moved from broadcast to local). This happens so that the

\* context data is not forwarded down tweetypie's backend rpc chains, which often result

\* in transitive calls back into tweetypie. This effectively creates single-hop marshalling.

\*/

object TweetypieContext {

// Bring Tweetypie permitted TwitterContext into scope

val TwitterContext: TwitterContext =

com.twitter.context.TwitterContext(com.twitter.tweetypie.TwitterContextPermit)

case class Ctx(via: String)

val Empty = Ctx("")

object Broadcast {

private[this] object Key extends Contexts.broadcast.Key[Ctx](id = Ctx.getClass.getName) {

override def marshal(value: Ctx): Buf =

Owned(value.via.getBytes(UTF\_8))

override def tryUnmarshal(buf: Buf): Try[Ctx] =

Try(Ctx(new String(Owned.extract(buf), UTF\_8)))

}

private[TweetypieContext] def current(): Option[Ctx] =

Contexts.broadcast.get(Key)

def currentOrElse(default: Ctx): Ctx =

current().getOrElse(default)

def letClear[T](f: => T): T =

Contexts.broadcast.letClear(Key)(f)

def let[T](ctx: Ctx)(f: => T): T =

if (Empty == ctx) {

letClear(f)

} else {

Contexts.broadcast.let(Key, ctx)(f)

}

// ctx has to be by name so we can re-evaluate it for every request (for usage in ServiceTwitter.scala)

def filter(ctx: => Ctx): Filter.TypeAgnostic =

new Filter.TypeAgnostic {

override def toFilter[Req, Rep]: Filter[Req, Rep, Req, Rep] =

(request: Req, service: Service[Req, Rep]) => Broadcast.let(ctx)(service(request))

}

}

object Local {

private[this] val Key =

new Contexts.local.Key[Ctx]

private[TweetypieContext] def let[T](ctx: Option[Ctx])(f: => T): T =

ctx match {

case Some(ctx) if ctx != Empty => Contexts.local.let(Key, ctx)(f)

case None => Contexts.local.letClear(Key)(f)

}

def current(): Option[Ctx] =

Contexts.local.get(Key)

def filter[Req, Rep]: SimpleFilter[Req, Rep] =

(request: Req, service: Service[Req, Rep]) => {

val ctx = Broadcast.current()

Broadcast.letClear(Local.let(ctx)(service(request)))

}

private[this] def clientAppIdToName(clientAppId: Long) =

GraphQlClientApplication.AllById.get(clientAppId).map(\_.name).getOrElse("nonTOO")

private[this] val pathRegexes: Seq[(Regex, String)] = Seq(

("timeline\_conversation\_.\*\_json".r, "timeline\_conversation\_\_slug\_\_json"),

("user\_timeline\_.\*\_json".r, "user\_timeline\_\_user\_\_json"),

("[0-9]{2,}".r, "\_id\_")

)

// `context.via` will either be a string like: "birdherd" or "birdherd:/1.1/statuses/show/123.json,

// depending on whether birdherd code was able to determine the path of the request.

private[this] def getViaAndPath(via: String): (String, Option[String]) =

via.split(":", 2) match {

case Array(via, path) =>

val sanitizedPath = path

.replace('/', '\_')

.replace('.', '\_')

// Apply each regex in turn

val normalizedPath = pathRegexes.foldLeft(sanitizedPath) {

case (path, (regex, replacement)) => regex.replaceAllIn(path, replacement)

}

(via, Some(normalizedPath))

case Array(via) => (via, None)

}

def trackStats[U](scopes: StatsReceiver\*): Unit =

for {

tweetypieCtx <- TweetypieContext.Local.current()

(via, pathOpt) = getViaAndPath(tweetypieCtx.via)

twitterCtx <- TwitterContext()

clientAppId <- twitterCtx.clientApplicationId

} yield {

val clientAppName = clientAppIdToName(clientAppId)

scopes.foreach { stats =>

val ctxStats = stats.scope("context")

val viaStats = ctxStats.scope("via", via)

viaStats.scope("all").counter("requests").incr()

val viaClientStats = viaStats.scope("by\_client", clientAppName)

viaClientStats.counter("requests").incr()

pathOpt.foreach { path =>

val viaPathStats = viaStats.scope("by\_path", path)

viaPathStats.counter("requests").incr()

}

}

}

}

}