package com.twitter.tweetypie.jiminy.tweetypie

import com.twitter.finagle.stats.CategorizingExceptionStatsHandler

import com.twitter.finagle.stats.Stat

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.incentives.jiminy.thriftscala.\_

import com.twitter.servo.util.FutureArrow

import com.twitter.servo.util.Gate

import com.twitter.stitch.Stitch

import com.twitter.strato.thrift.ScroogeConvImplicits.\_

import com.twitter.strato.client.{Client => StratoClient}

import com.twitter.tweetypie.core.TweetCreateFailure

import com.twitter.util.Future

import com.twitter.util.Return

import com.twitter.util.Throw

case class NudgeBuilderRequest(

text: String,

inReplyToTweetId: Option[NudgeBuilder.TweetId],

conversationId: Option[NudgeBuilder.TweetId],

hasQuotedTweet: Boolean,

nudgeOptions: Option[CreateTweetNudgeOptions],

tweetId: Option[NudgeBuilder.TweetId])

trait NudgeBuilder extends FutureArrow[NudgeBuilderRequest, Unit] {

/\*\*

\* Check whether the user should receive a nudge instead of creating

\* the Tweet. If nudgeOptions is None, then no nudge check will be

\* performed.

\*

\* @return a Future.exception containing a [[TweetCreateFailure]] if the

\* user should be nudged, or Future.Unit if the user should not be

\* nudged.

\*/

def apply(

request: NudgeBuilderRequest

): Future[Unit]

}

object NudgeBuilder {

type Type = FutureArrow[NudgeBuilderRequest, Unit]

type TweetId = Long

// darkTrafficCreateNudgeOptions ensure that our dark traffic sends a request that will

// accurately test the Jiminy backend. in this case, we specify that we want checks for all

// possible nudge types

private[this] val darkTrafficCreateNudgeOptions = Some(

CreateTweetNudgeOptions(

requestedNudgeTypes = Some(

Set(

TweetNudgeType.PotentiallyToxicTweet,

TweetNudgeType.ReviseOrMute,

TweetNudgeType.ReviseOrHideThenBlock,

TweetNudgeType.ReviseOrBlock

)

)

)

)

private[this] def mkJiminyRequest(

request: NudgeBuilderRequest,

isDarkRequest: Boolean = false

): CreateTweetNudgeRequest = {

val tweetType =

if (request.inReplyToTweetId.nonEmpty) TweetType.Reply

else if (request.hasQuotedTweet) TweetType.QuoteTweet

else TweetType.OriginalTweet

CreateTweetNudgeRequest(

tweetText = request.text,

tweetType = tweetType,

inReplyToTweetId = request.inReplyToTweetId,

conversationId = request.conversationId,

createTweetNudgeOptions =

if (isDarkRequest) darkTrafficCreateNudgeOptions else request.nudgeOptions,

tweetId = request.tweetId

)

}

/\*\*

\* NudgeBuilder implemented by calling the strato column `incentives/createNudge`.

\*

\* Stats recorded:

\* - latency\_ms: Latency histogram (also implicitly number of

\* invocations). This is counted only in the case that a nudge

\* check was requested (`nudgeOptions` is non-empty)

\*

\* - nudge: The nudge check succeeded and a nudge was created.

\*

\* - no\_nudge: The nudge check succeeded, but no nudge was created.

\*

\* - failures: Calling strato to create a nudge failed. Broken out

\* by exception.

\*/

def apply(

nudgeArrow: FutureArrow[CreateTweetNudgeRequest, CreateTweetNudgeResponse],

enableDarkTraffic: Gate[Unit],

stats: StatsReceiver

): NudgeBuilder = {

new NudgeBuilder {

private[this] val nudgeLatencyStat = stats.stat("latency\_ms")

private[this] val nudgeCounter = stats.counter("nudge")

private[this] val noNudgeCounter = stats.counter("no\_nudge")

private[this] val darkRequestCounter = stats.counter("dark\_request")

private[this] val nudgeExceptionHandler = new CategorizingExceptionStatsHandler

override def apply(

request: NudgeBuilderRequest

): Future[Unit] =

request.nudgeOptions match {

case None =>

if (enableDarkTraffic()) {

darkRequestCounter.incr()

Stat

.timeFuture(nudgeLatencyStat) {

nudgeArrow(mkJiminyRequest(request, isDarkRequest = true))

}

.transform { \_ =>

// ignore the response since it is a dark request

Future.Done

}

} else {

Future.Done

}

case Some(\_) =>

Stat

.timeFuture(nudgeLatencyStat) {

nudgeArrow(mkJiminyRequest(request))

}

.transform {

case Throw(e) =>

nudgeExceptionHandler.record(stats, e)

// If we failed to invoke the nudge column, then

// just continue on with the Tweet creation.

Future.Done

case Return(CreateTweetNudgeResponse(Some(nudge))) =>

nudgeCounter.incr()

Future.exception(TweetCreateFailure.Nudged(nudge = nudge))

case Return(CreateTweetNudgeResponse(None)) =>

noNudgeCounter.incr()

Future.Done

}

}

}

}

def apply(

strato: StratoClient,

enableDarkTraffic: Gate[Unit],

stats: StatsReceiver

): NudgeBuilder = {

val executer =

strato.executer[CreateTweetNudgeRequest, CreateTweetNudgeResponse](

"incentives/createTweetNudge")

val nudgeArrow: FutureArrow[CreateTweetNudgeRequest, CreateTweetNudgeResponse] = { req =>

Stitch.run(executer.execute(req))

}

apply(nudgeArrow, enableDarkTraffic, stats)

}

}