package com.twitter.product\_mixer.core.service.gate\_executor

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.product\_mixer.core.functional\_component.gate.BaseGate

import com.twitter.product\_mixer.core.functional\_component.gate.GateResult

import com.twitter.product\_mixer.core.pipeline.PipelineQuery

import com.twitter.product\_mixer.core.service.Executor

import com.twitter.stitch.Arrow

import com.twitter.stitch.Arrow.Iso

import com.twitter.util.Return

import com.twitter.util.Throw

import javax.inject.Inject

import javax.inject.Singleton

import scala.collection.immutable.Queue

/\*\*

\* A GateExecutor takes a Seq[Gate], executes them all sequentially, and

\* determines a final Continue or Stop decision.

\*/

@Singleton

class GateExecutor @Inject() (override val statsReceiver: StatsReceiver) extends Executor {

private val Continue = "continue"

private val Skipped = "skipped"

private val Stop = "stop"

def arrow[Query <: PipelineQuery](

gates: Seq[BaseGate[Query]],

context: Executor.Context

): Arrow[Query, GateExecutorResult] = {

val gateArrows = gates.map(getIsoArrowForGate(\_, context))

val combinedArrow = isoArrowsSequentially(gateArrows)

Arrow

.map { query: Query => (query, GateExecutorResult(Queue.empty)) }

.andThen(combinedArrow)

.map {

case (\_, gateExecutorResult) =>

// materialize the Queue into a List for faster future iterations

GateExecutorResult(gateExecutorResult.individualGateResults.toList)

}

}

/\*\*

\* Each gate is transformed into a Iso Arrow over (Quest, List[GatewayResult]).

\*

\* This arrow:

\* - Adapts the input and output types of the underlying Gate arrow (an [[Iso[(Query, QueryResult)]])

\* - throws a [[StoppedGateException]] if [[GateResult.continue]] is false

\* - if its not false, prepends the current results to the [[GateExecutorResult.individualGateResults]] list

\*/

private def getIsoArrowForGate[Query <: PipelineQuery](

gate: BaseGate[Query],

context: Executor.Context

): Iso[(Query, GateExecutorResult)] = {

val broadcastStatsReceiver =

Executor.broadcastStatsReceiver(context, gate.identifier, statsReceiver)

val continueCounter = broadcastStatsReceiver.counter(Continue)

val skippedCounter = broadcastStatsReceiver.counter(Skipped)

val stopCounter = broadcastStatsReceiver.counter(Stop)

val observedArrow = wrapComponentWithExecutorBookkeeping(

context,

gate.identifier,

onSuccess = { gateResult: GateResult =>

gateResult match {

case GateResult.Continue => continueCounter.incr()

case GateResult.Skipped => skippedCounter.incr()

case GateResult.Stop => stopCounter.incr()

}

}

)(gate.arrow)

val inputAdapted: Arrow[(Query, GateExecutorResult), GateResult] =

Arrow

.map[(Query, GateExecutorResult), Query] { case (query, \_) => query }

.andThen(observedArrow)

val zipped = Arrow.zipWithArg(inputAdapted)

// at each step, the current `GateExecutorResult.continue` value is correct for all already run gates

val withStoppedGatesAsExceptions = zipped.map {

case ((query, previousResults), currentResult) if currentResult.continue =>

Return(

(

query,

GateExecutorResult(

previousResults.individualGateResults :+ ExecutedGateResult(

gate.identifier,

currentResult))

))

case \_ => Throw(StoppedGateException(gate.identifier))

}.lowerFromTry

/\*\*

\* we gather stats before converting closed gates to exceptions because a closed gate

\* isn't a failure for the gate, its a normal behavior

\* but we do want to remap the the [[StoppedGateException]] created because the [[BaseGate]] is closed

\* to the correct [[com.twitter.product\_mixer.core.pipeline.pipeline\_failure.PipelineFailure]],

\* so we remap with [[wrapWithErrorHandling]]

\*/

wrapWithErrorHandling(context, gate.identifier)(withStoppedGatesAsExceptions)

}

}