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

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.product\_mixer.core.model.common.identifier.ComponentIdentifier

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

import com.twitter.product\_mixer.core.pipeline.pipeline\_failure.InvalidPipelineSelected

import com.twitter.product\_mixer.core.pipeline.pipeline\_failure.PipelineFailure

import com.twitter.product\_mixer.core.quality\_factor.QualityFactorObserver

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

import com.twitter.stitch.Arrow

import com.twitter.util.logging.Logging

import javax.inject.Inject

import javax.inject.Singleton

/\*\*

\* PipelineExecutor executes a single pipeline (of any type)

\* It does not currently support fail open/closed policies like CandidatePipelineExecutor does

\* In the future, maybe they can be merged.

\*/

case class PipelineExecutorRequest[Query](query: Query, pipelineIdentifier: ComponentIdentifier)

@Singleton

class PipelineExecutor @Inject() (override val statsReceiver: StatsReceiver)

extends Executor

with Logging {

def arrow[Query, ResultType](

pipelineByIdentifier: Map[ComponentIdentifier, Pipeline[Query, ResultType]],

qualityFactorObserverByPipeline: Map[ComponentIdentifier, QualityFactorObserver],

context: Executor.Context

): Arrow[PipelineExecutorRequest[Query], PipelineExecutorResult[ResultType]] = {

val wrappedPipelineArrowsByIdentifier = pipelineByIdentifier.mapValues { pipeline =>

wrapPipelineWithExecutorBookkeeping(

context,

pipeline.identifier,

qualityFactorObserverByPipeline.get(pipeline.identifier))(pipeline.arrow)

}

val appliedPipelineArrow = Arrow

.identity[PipelineExecutorRequest[Query]]

.map {

case PipelineExecutorRequest(query, pipelineIdentifier) =>

val pipeline = wrappedPipelineArrowsByIdentifier.getOrElse(

pipelineIdentifier,

// throwing instead of returning a `Throw(\_)` and then `.lowerFromTry` because this is an exceptional case and we want to emphasize that by explicitly throwing

// this case should never happen since this is checked in the `PipelineSelectorExecutor` but we check it anyway

throw PipelineFailure(

InvalidPipelineSelected,

s"${context.componentStack.peek} attempted to execute $pipelineIdentifier",

// the `componentStack` includes the missing pipeline so it can show up in metrics easier

componentStack = Some(context.componentStack.push(pipelineIdentifier))

)

)

(pipeline, query)

}

// less efficient than an `andThen` but since we dispatch this dynamically we need to use either `applyArrow` or `flatMap` and this is the better of those options

.applyArrow

.map(PipelineExecutorResult(\_))

// no additional error handling needed since we populate the component stack above already

appliedPipelineArrow

}

}