package com.twitter.product\_mixer.core.service.candidate\_pipeline\_executor

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.product\_mixer.core.feature.featuremap.FeatureMap

import com.twitter.product\_mixer.core.feature.featuremap.FeatureMapBuilder

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

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

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

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

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

import com.twitter.product\_mixer.core.pipeline.candidate.CandidatePipeline

import com.twitter.product\_mixer.core.pipeline.candidate.CandidatePipelineResult

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

@Singleton

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

extends Executor

with Logging {

def arrow[Query <: PipelineQuery](

candidatePipelines: Seq[CandidatePipeline[Query]],

defaultFailOpenPolicy: FailOpenPolicy,

failOpenPolicies: Map[CandidatePipelineIdentifier, FailOpenPolicy],

qualityFactorObserverByPipeline: Map[ComponentIdentifier, QualityFactorObserver],

context: Executor.Context

): Arrow[CandidatePipeline.Inputs[Query], CandidatePipelineExecutorResult] = {

// Get the `.arrow` of each Candidate Pipeline, and wrap it in a ResultObserver

val observedArrows: Seq[Arrow[CandidatePipeline.Inputs[Query], CandidatePipelineResult]] =

candidatePipelines.map { pipeline =>

wrapPipelineWithExecutorBookkeeping(

context = context,

currentComponentIdentifier = pipeline.identifier,

qualityFactorObserver = qualityFactorObserverByPipeline.get(pipeline.identifier),

failOpenPolicy = failOpenPolicies.getOrElse(pipeline.identifier, defaultFailOpenPolicy)

)(pipeline.arrow)

}

// Collect the results from all the candidate pipelines together

Arrow.zipWithArg(Arrow.collect(observedArrows)).map {

case (input: CandidatePipeline.Inputs[Query], results: Seq[CandidatePipelineResult]) =>

val candidateWithDetails = results.flatMap(\_.result.getOrElse(Seq.empty))

val previousCandidateWithDetails = input.query.features

.map(\_.getOrElse(CandidatePipelineResults, Seq.empty))

.getOrElse(Seq.empty)

val featureMapWithCandidates = FeatureMapBuilder()

.add(CandidatePipelineResults, previousCandidateWithDetails ++ candidateWithDetails)

.build()

// Merge the query feature hydrator and candidate source query features back in. While this

// is done internally in the pipeline, we have to pass it back since we don't expose the

// updated pipeline query today.

val queryFeatureHydratorFeatureMaps =

results

.flatMap(result => Seq(result.queryFeatures, result.queryFeaturesPhase2))

.collect { case Some(result) => result.featureMap }

val asyncFeatureHydratorFeatureMaps =

results

.flatMap(\_.asyncFeatureHydrationResults)

.flatMap(\_.featureMapsByStep.values)

val candidateSourceFeatureMaps =

results

.flatMap(\_.candidateSourceResult)

.map(\_.candidateSourceFeatureMap)

val featureMaps =

(featureMapWithCandidates +: queryFeatureHydratorFeatureMaps) ++ asyncFeatureHydratorFeatureMaps ++ candidateSourceFeatureMaps

val mergedFeatureMap = FeatureMap.merge(featureMaps)

CandidatePipelineExecutorResult(

candidatePipelineResults = results,

queryFeatureMap = mergedFeatureMap)

}

}

}