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

import com.twitter.finagle.stats.StatsReceiver

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

import com.twitter.product\_mixer.core.model.common.UniversalNoun

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

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

import com.twitter.product\_mixer.core.model.common.presentation.ItemCandidateWithDetails

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

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

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

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

import com.twitter.product\_mixer.core.pipeline.scoring.ScoringPipeline

import com.twitter.product\_mixer.core.pipeline.scoring.ScoringPipelineResult

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

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

import com.twitter.product\_mixer.core.service.scoring\_pipeline\_executor.ScoringPipelineExecutor.ScoringPipelineState

import com.twitter.stitch.Arrow

import com.twitter.stitch.Arrow.Iso

import com.twitter.util.logging.Logging

import javax.inject.Inject

import javax.inject.Singleton

import scala.collection.immutable.Queue

@Singleton

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

extends Executor

with Logging {

def arrow[Query <: PipelineQuery, Candidate <: UniversalNoun[Any]](

pipelines: Seq[ScoringPipeline[Query, Candidate]],

context: Executor.Context,

defaultFailOpenPolicy: FailOpenPolicy,

failOpenPolicies: Map[ScoringPipelineIdentifier, FailOpenPolicy],

qualityFactorObserverByPipeline: Map[ComponentIdentifier, QualityFactorObserver],

): Arrow[ScoringPipelineExecutor.Inputs[Query], ScoringPipelineExecutorResult[Candidate]] = {

val scoringPipelineArrows = pipelines.map { pipeline =>

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

val qualityFactorObserver = qualityFactorObserverByPipeline.get(pipeline.identifier)

getIsoArrowForScoringPipeline(

pipeline,

context,

failOpenPolicy,

qualityFactorObserver

)

}

val combinedArrow = isoArrowsSequentially(scoringPipelineArrows)

Arrow

.map[ScoringPipelineExecutor.Inputs[Query], ScoringPipelineState[Query, Candidate]] {

case input =>

ScoringPipelineState(

input.query,

input.itemCandidatesWithDetails,

ScoringPipelineExecutorResult(input.itemCandidatesWithDetails, Queue.empty))

}.flatMapArrow(combinedArrow).map { state =>

state.executorResult.copy(individualPipelineResults =

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

state.executorResult.individualPipelineResults.toList)

}

}

private def getIsoArrowForScoringPipeline[

Query <: PipelineQuery,

Candidate <: UniversalNoun[Any]

](

pipeline: ScoringPipeline[Query, Candidate],

context: Executor.Context,

failOpenPolicy: FailOpenPolicy,

qualityFactorObserver: Option[QualityFactorObserver]

): Iso[ScoringPipelineState[Query, Candidate]] = {

val pipelineArrow = Arrow

.map[ScoringPipelineState[Query, Candidate], ScoringPipeline.Inputs[Query]] { state =>

ScoringPipeline.Inputs(state.query, state.allCandidates)

}.flatMapArrow(pipeline.arrow)

val observedArrow = wrapPipelineWithExecutorBookkeeping(

context,

pipeline.identifier,

qualityFactorObserver,

failOpenPolicy)(pipelineArrow)

Arrow

.zipWithArg(

observedArrow

).map {

case (

scoringPipelinesState: ScoringPipelineState[Query, Candidate],

scoringPipelineResult: ScoringPipelineResult[Candidate]) =>

val updatedCandidates: Seq[ItemCandidateWithDetails] =

mkUpdatedCandidates(pipeline.identifier, scoringPipelinesState, scoringPipelineResult)

ScoringPipelineState(

scoringPipelinesState.query,

updatedCandidates,

scoringPipelinesState.executorResult

.copy(

updatedCandidates,

scoringPipelinesState.executorResult.individualPipelineResults :+ scoringPipelineResult)

)

}

}

private def mkUpdatedCandidates[Query <: PipelineQuery, Candidate <: UniversalNoun[Any]](

scoringPipelineIdentifier: ScoringPipelineIdentifier,

scoringPipelinesState: ScoringPipelineState[Query, Candidate],

scoringPipelineResult: ScoringPipelineResult[Candidate]

): Seq[ItemCandidateWithDetails] = {

if (scoringPipelineResult.failure.isEmpty) {

/\*\*

\* It's important that we map back from which actual item candidate was scored by looking

\* at the selector results. This is to defend against the same candidate being selected

\* from two different candidate pipelines. If one is selected and the other isn't, we

\* should only score the selected one. If both are selected and each is scored differently

\* we should get the right score for each.

\*/

val selectedItemCandidates: Seq[ItemCandidateWithDetails] =

scoringPipelineResult.selectorResults

.getOrElse(throw PipelineFailure(

IllegalStateFailure,

s"Missing Selector Results in Scoring Pipeline $scoringPipelineIdentifier")).selectedCandidates.collect {

case itemCandidateWithDetails: ItemCandidateWithDetails =>

itemCandidateWithDetails

}

val scoredFeatureMaps: Seq[FeatureMap] = scoringPipelineResult.result

.getOrElse(Seq.empty).map(\_.features)

if (scoredFeatureMaps.isEmpty) {

// It's possible that all Scorers are [[Conditionally]] off. In this case, we return empty

// and don't validate the list size since this is done in the hydrator/scorer executor.

scoringPipelinesState.allCandidates

} else if (selectedItemCandidates.length != scoredFeatureMaps.length) {

// The length of the inputted candidates should always match the returned feature map, unless

throw PipelineFailure(

IllegalStateFailure,

s"Missing configured scorer result, length of scorer results does not match the length of selected candidates")

} else {

/\* Zip the selected item candidate seq back to the scored feature maps, this works

\* because the scored results will always have the same number of elements returned

\* and it should match the same order. We then loop through all candidates because the

\* expectation is to always keep the result since a subsequent scoring pipeline can score a

\* candidate that the current one did not. We only update the feature map of the candidate

\* if it was selected and scored.

\*/

val selectedItemCandidateToScorerMap: Map[ItemCandidateWithDetails, FeatureMap] =

selectedItemCandidates.zip(scoredFeatureMaps).toMap

scoringPipelinesState.allCandidates.map { itemCandidateWithDetails =>

selectedItemCandidateToScorerMap.get(itemCandidateWithDetails) match {

case Some(scorerResult) =>

itemCandidateWithDetails.copy(features =

itemCandidateWithDetails.features ++ scorerResult)

case None => itemCandidateWithDetails

}

}

}

} else {

// If the underlying scoring pipeline has failed open, just keep the existing candidates

scoringPipelinesState.allCandidates

}

}

}

object ScoringPipelineExecutor {

private case class ScoringPipelineState[Query <: PipelineQuery, Candidate <: UniversalNoun[Any]](

query: Query,

allCandidates: Seq[ItemCandidateWithDetails],

executorResult: ScoringPipelineExecutorResult[Candidate])

case class Inputs[Query <: PipelineQuery](

query: Query,

itemCandidatesWithDetails: Seq[ItemCandidateWithDetails])

}