package com.twitter.product\_mixer.core.service.candidate\_feature\_hydrator\_executor

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.product\_mixer.core.feature.Feature

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

import com.twitter.product\_mixer.core.functional\_component.feature\_hydrator.BaseBulkCandidateFeatureHydrator

import com.twitter.product\_mixer.core.functional\_component.feature\_hydrator.BaseCandidateFeatureHydrator

import com.twitter.product\_mixer.core.functional\_component.feature\_hydrator.CandidateFeatureHydrator

import com.twitter.product\_mixer.core.functional\_component.feature\_hydrator.HydratorCandidateResult

import com.twitter.product\_mixer.core.functional\_component.feature\_hydrator.featurestorev1.FeatureStoreV1CandidateFeatureHydrator

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

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

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

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

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

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

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

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

import com.twitter.product\_mixer.core.service.candidate\_feature\_hydrator\_executor.CandidateFeatureHydratorExecutor.Inputs

import com.twitter.product\_mixer.core.service.feature\_hydrator\_observer.FeatureHydratorObserver

import com.twitter.stitch.Arrow

import com.twitter.util.Try

import javax.inject.Inject

import javax.inject.Singleton

@Singleton

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

extends Executor {

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

hydrators: Seq[BaseCandidateFeatureHydrator[Query, Result, \_]],

context: Executor.Context

): Arrow[

Inputs[Query, Result],

CandidateFeatureHydratorExecutorResult[

Result

]

] = {

val observer = new FeatureHydratorObserver(statsReceiver, hydrators, context)

val candidateFeatureHydratorExecutorResults: Seq[Arrow[

Inputs[Query, Result],

CandidateFeatureHydratorExecutorResult[Result]

]] = hydrators.map(getCandidateHydratorArrow(\_, context, observer))

val runHydrators = Arrow.collect(candidateFeatureHydratorExecutorResults).map {

candidateFeatureHydratorExecutorResult: Seq[CandidateFeatureHydratorExecutorResult[Result]] =>

candidateFeatureHydratorExecutorResult.foldLeft(

CandidateFeatureHydratorExecutorResult[Result](

Seq.empty,

Map.empty

)

) { (accumulator, additionalResult) =>

// accumulator.results and additionalResults.results are either the same length or one may be empty

// checks in each Hydrator's Arrow implementation ensure the ordering and length are correct

val mergedFeatureMaps =

if (accumulator.results.length == additionalResult.results.length) {

// merge if there are results for both and they are the same size

// also handles both being empty

accumulator.results.zip(additionalResult.results).map {

case (accumulatedScoredCandidate, resultScoredCandidate) =>

val updatedFeatureMap =

accumulatedScoredCandidate.features ++ resultScoredCandidate.features

HydratorCandidateResult(resultScoredCandidate.candidate, updatedFeatureMap)

}

} else if (accumulator.results.isEmpty) {

// accumulator is empty (the initial case) so use additional results

additionalResult.results

} else {

// empty results but non-empty accumulator due to Hydrator being turned off so use accumulator results

accumulator.results

}

CandidateFeatureHydratorExecutorResult(

mergedFeatureMaps,

accumulator.individualFeatureHydratorResults ++ additionalResult.individualFeatureHydratorResults

)

}

}

Arrow.ifelse[Inputs[Query, Result], CandidateFeatureHydratorExecutorResult[Result]](

\_.candidates.nonEmpty,

runHydrators,

Arrow.value(CandidateFeatureHydratorExecutorResult(Seq.empty, Map.empty)))

}

/\*\* @note the returned [[Arrow]] must have a result for every candidate passed into it in the same order OR a completely empty result \*/

private def getCandidateHydratorArrow[Query <: PipelineQuery, Result <: UniversalNoun[Any]](

hydrator: BaseCandidateFeatureHydrator[Query, Result, \_],

context: Executor.Context,

candidateFeatureHydratorObserver: FeatureHydratorObserver

): Arrow[

Inputs[Query, Result],

CandidateFeatureHydratorExecutorResult[Result]

] = {

val componentExecutorContext = context.pushToComponentStack(hydrator.identifier)

val validateFeatureMapFn: FeatureMap => FeatureMap =

hydrator match {

// Feature store candidate hydrators store the resulting PredictionRecords and

// not the features, so we cannot validate the same way

case \_: FeatureStoreV1CandidateFeatureHydrator[Query, Result] =>

identity

case \_ =>

validateFeatureMap(

hydrator.features.asInstanceOf[Set[Feature[\_, \_]]],

\_,

componentExecutorContext)

}

val hydratorBaseArrow = hydrator match {

case hydrator: CandidateFeatureHydrator[Query, Result] =>

singleCandidateHydratorArrow(

hydrator,

validateFeatureMapFn,

componentExecutorContext,

parentContext = context)

case hydrator: BaseBulkCandidateFeatureHydrator[Query, Result, \_] =>

bulkCandidateHydratorArrow(

hydrator,

validateFeatureMapFn,

componentExecutorContext,

parentContext = context)

}

val candidateFeatureHydratorArrow =

Arrow

.zipWithArg(hydratorBaseArrow)

.map {

case (

arg: CandidateFeatureHydratorExecutor.Inputs[Query, Result],

featureMapSeq: Seq[FeatureMap]) =>

val candidates = arg.candidates.map(\_.candidate)

candidateFeatureHydratorObserver.observeFeatureSuccessAndFailures(

hydrator,

featureMapSeq)

// Build a map from candidate to FeatureMap

val candidateAndFeatureMaps = if (candidates.size == featureMapSeq.size) {

candidates.zip(featureMapSeq).map {

case (candidate, featureMap) => HydratorCandidateResult(candidate, featureMap)

}

} else {

throw PipelineFailure(

MisconfiguredFeatureMapFailure,

s"Unexpected response length from ${hydrator.identifier}, ensure hydrator returns feature map for all candidates")

}

val individualFeatureHydratorFeatureMaps =

Map(hydrator.identifier -> IndividualFeatureHydratorResult(candidateAndFeatureMaps))

CandidateFeatureHydratorExecutorResult(

candidateAndFeatureMaps,

individualFeatureHydratorFeatureMaps)

}

val conditionallyRunArrow = hydrator match {

case hydrator: BaseCandidateFeatureHydrator[Query, Result, \_] with Conditionally[

Query @unchecked

] =>

Arrow.ifelse[Inputs[Query, Result], CandidateFeatureHydratorExecutorResult[Result]](

{ case Inputs(query: Query @unchecked, \_) => hydrator.onlyIf(query) },

candidateFeatureHydratorArrow,

Arrow.value(

CandidateFeatureHydratorExecutorResult(

Seq.empty,

Map(hydrator.identifier -> FeatureHydratorDisabled[Result]())

))

)

case \_ => candidateFeatureHydratorArrow

}

wrapWithErrorHandling(context, hydrator.identifier)(conditionallyRunArrow)

}

private def singleCandidateHydratorArrow[Query <: PipelineQuery, Result <: UniversalNoun[Any]](

hydrator: CandidateFeatureHydrator[Query, Result],

validateFeatureMap: FeatureMap => FeatureMap,

componentContext: Context,

parentContext: Context

): Arrow[Inputs[Query, Result], Seq[FeatureMap]] = {

val inputTransformer = Arrow

.map { inputs: Inputs[Query, Result] =>

inputs.candidates.map { candidate =>

(inputs.query, candidate.candidate, candidate.features)

}

}

val hydratorArrow = Arrow

.flatMap[(Query, Result, FeatureMap), FeatureMap] {

case (query, candidate, featureMap) =>

hydrator.apply(query, candidate, featureMap)

}

// validate before observing so validation failures are caught in the metrics

val hydratorArrowWithValidation = hydratorArrow.map(validateFeatureMap)

// no tracing here since per-Component spans is overkill

val observedArrow =

wrapPerCandidateComponentWithExecutorBookkeepingWithoutTracing(

parentContext,

hydrator.identifier

)(hydratorArrowWithValidation)

// only handle non-validation failures

val liftNonValidationFailuresToFailedFeatures = Arrow.handle[FeatureMap, FeatureMap] {

case NotAMisconfiguredFeatureMapFailure(e) =>

featureMapWithFailuresForFeatures(hydrator.features, e, componentContext)

}

wrapComponentsWithTracingOnly(parentContext, hydrator.identifier)(

inputTransformer.andThen(

Arrow.sequence(observedArrow.andThen(liftNonValidationFailuresToFailedFeatures))

)

)

}

private def bulkCandidateHydratorArrow[Query <: PipelineQuery, Result <: UniversalNoun[Any]](

hydrator: BaseBulkCandidateFeatureHydrator[Query, Result, \_],

validateFeatureMap: FeatureMap => FeatureMap,

componentContext: Context,

parentContext: Context

): Arrow[Inputs[Query, Result], Seq[FeatureMap]] = {

val hydratorArrow: Arrow[Inputs[Query, Result], Seq[FeatureMap]] =

Arrow.flatMap { inputs =>

hydrator.apply(inputs.query, inputs.candidates)

}

val validationArrow: Arrow[(Inputs[Query, Result], Seq[FeatureMap]), Seq[FeatureMap]] = Arrow

.map[(Inputs[Query, Result], Seq[FeatureMap]), Seq[FeatureMap]] {

case (inputs, results) =>

// For bulk APIs, this ensures no candidates are omitted and also ensures the order is preserved.

if (inputs.candidates.length != results.length) {

throw PipelineFailure(

MisconfiguredFeatureMapFailure,

s"Unexpected response from ${hydrator.identifier}, ensure hydrator returns features for all candidates. Missing results for ${inputs.candidates.length - results.length} candidates"

)

}

results.map(validateFeatureMap)

}

// validate before observing so validation failures are caught in the metrics

val hydratorArrowWithValidation: Arrow[Inputs[Query, Result], Seq[FeatureMap]] =

Arrow.zipWithArg(hydratorArrow).andThen(validationArrow)

val observedArrow =

wrapComponentWithExecutorBookkeeping(parentContext, hydrator.identifier)(

hydratorArrowWithValidation)

// only handle non-validation failures

val liftNonValidationFailuresToFailedFeatures =

Arrow.map[(Inputs[Query, Result], Try[Seq[FeatureMap]]), Try[Seq[FeatureMap]]] {

case (inputs, resultTry) =>

resultTry.handle {

case NotAMisconfiguredFeatureMapFailure(e) =>

val errorFeatureMap =

featureMapWithFailuresForFeatures(

hydrator.features.asInstanceOf[Set[Feature[\_, \_]]],

e,

componentContext)

inputs.candidates.map(\_ => errorFeatureMap)

}

}

Arrow

.zipWithArg(observedArrow.liftToTry)

.andThen(liftNonValidationFailuresToFailedFeatures)

.lowerFromTry

}

}

object CandidateFeatureHydratorExecutor {

case class Inputs[+Query <: PipelineQuery, Candidate <: UniversalNoun[Any]](

query: Query,

candidates: Seq[CandidateWithFeatures[Candidate]])

}