package com.twitter.product\_mixer.core.pipeline.recommendation

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.util.logging.Logging

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

import com.twitter.product\_mixer.core.feature.featuremap.asyncfeaturemap.AsyncFeatureMap

import com.twitter.product\_mixer.core.functional\_component.common.alert.Alert

import com.twitter.product\_mixer.core.functional\_component.decorator.CandidateDecorator

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

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

import com.twitter.product\_mixer.core.functional\_component.filter.Filter

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

import com.twitter.product\_mixer.core.functional\_component.premarshaller.DomainMarshaller

import com.twitter.product\_mixer.core.functional\_component.selector.Selector

import com.twitter.product\_mixer.core.functional\_component.side\_effect.PipelineResultSideEffect

import com.twitter.product\_mixer.core.functional\_component.marshaller.TransportMarshaller

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

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

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

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.model.common.identifier.ComponentIdentifierStack

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

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

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

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

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

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

import com.twitter.product\_mixer.core.model.marshalling.HasMarshalling

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

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

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

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.CandidatePipelineBuilderFactory

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

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

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

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

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

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

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

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

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

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

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

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

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

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

import com.twitter.product\_mixer.core.service.async\_feature\_map\_executor.AsyncFeatureMapExecutor

import com.twitter.product\_mixer.core.service.async\_feature\_map\_executor.AsyncFeatureMapExecutorResults

import com.twitter.product\_mixer.core.service.candidate\_decorator\_executor.CandidateDecoratorExecutor

import com.twitter.product\_mixer.core.service.candidate\_decorator\_executor.CandidateDecoratorExecutorResult

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

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

import com.twitter.product\_mixer.core.service.candidate\_pipeline\_executor.CandidatePipelineExecutor

import com.twitter.product\_mixer.core.service.candidate\_pipeline\_executor.CandidatePipelineExecutorResult

import com.twitter.product\_mixer.core.service.domain\_marshaller\_executor.DomainMarshallerExecutor

import com.twitter.product\_mixer.core.service.filter\_executor.FilterExecutor

import com.twitter.product\_mixer.core.service.filter\_executor.FilterExecutorResult

import com.twitter.product\_mixer.core.service.gate\_executor.GateExecutor

import com.twitter.product\_mixer.core.service.gate\_executor.GateExecutorResult

import com.twitter.product\_mixer.core.service.gate\_executor.StoppedGateException

import com.twitter.product\_mixer.core.service.pipeline\_result\_side\_effect\_executor.PipelineResultSideEffectExecutor

import com.twitter.product\_mixer.core.service.quality\_factor\_executor.QualityFactorExecutorResult

import com.twitter.product\_mixer.core.service.query\_feature\_hydrator\_executor.QueryFeatureHydratorExecutor

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

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

import com.twitter.product\_mixer.core.service.selector\_executor.SelectorExecutor

import com.twitter.product\_mixer.core.service.selector\_executor.SelectorExecutorResult

import com.twitter.product\_mixer.core.service.transport\_marshaller\_executor.TransportMarshallerExecutor

import com.twitter.stitch.Arrow

/\*\*

\* RecommendationPipelineBuilder builds [[RecommendationPipeline]]s from [[RecommendationPipelineConfig]]s.

\*

\* You should inject a [[RecommendationPipelineBuilderFactory]] and call `.get` to build these.

\*

\* @see [[RecommendationPipelineConfig]] for the description of the type parameters.

\*

\* @note Almost a mirror of MixerPipelineBuilder

\*/

class RecommendationPipelineBuilder[

Query <: PipelineQuery,

Candidate <: UniversalNoun[Any],

DomainResultType <: HasMarshalling,

Result

](

candidatePipelineExecutor: CandidatePipelineExecutor,

gateExecutor: GateExecutor,

selectorExecutor: SelectorExecutor,

queryFeatureHydratorExecutor: QueryFeatureHydratorExecutor,

asyncFeatureMapExecutor: AsyncFeatureMapExecutor,

candidateFeatureHydratorExecutor: CandidateFeatureHydratorExecutor,

filterExecutor: FilterExecutor,

scoringPipelineExecutor: ScoringPipelineExecutor,

candidateDecoratorExecutor: CandidateDecoratorExecutor,

domainMarshallerExecutor: DomainMarshallerExecutor,

transportMarshallerExecutor: TransportMarshallerExecutor,

pipelineResultSideEffectExecutor: PipelineResultSideEffectExecutor,

candidatePipelineBuilderFactory: CandidatePipelineBuilderFactory,

scoringPipelineBuilderFactory: ScoringPipelineBuilderFactory,

override val statsReceiver: StatsReceiver)

extends PipelineBuilder[Query]

with Logging {

override type UnderlyingResultType = Result

override type PipelineResultType = RecommendationPipelineResult[Candidate, Result]

def qualityFactorStep(

qualityFactorStatus: QualityFactorStatus

): Step[Query, QualityFactorExecutorResult] =

new Step[Query, QualityFactorExecutorResult] {

override def identifier: PipelineStepIdentifier =

RecommendationPipelineConfig.qualityFactorStep

override def executorArrow: Arrow[Query, QualityFactorExecutorResult] =

Arrow

.map[Query, QualityFactorExecutorResult] { \_ =>

QualityFactorExecutorResult(

pipelineQualityFactors =

qualityFactorStatus.qualityFactorByPipeline.mapValues(\_.currentValue)

)

}

override def inputAdaptor(

query: Query,

previousResult: RecommendationPipelineResult[Candidate, Result]

): Query = query

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: QualityFactorExecutorResult

): RecommendationPipelineResult[Candidate, Result] =

previousPipelineResult.copy(qualityFactorResult = Some(executorResult))

override def queryUpdater(

query: Query,

executorResult: QualityFactorExecutorResult

): Query = {

query match {

case queryWithQualityFactor: HasQualityFactorStatus =>

queryWithQualityFactor

.withQualityFactorStatus(

queryWithQualityFactor.qualityFactorStatus.getOrElse(QualityFactorStatus.empty) ++

qualityFactorStatus

).asInstanceOf[Query]

case \_ =>

query

}

}

}

def gatesStep(

gates: Seq[Gate[Query]],

context: Executor.Context

): Step[Query, GateExecutorResult] = new Step[Query, GateExecutorResult] {

override def identifier: PipelineStepIdentifier = RecommendationPipelineConfig.gatesStep

override def executorArrow: Arrow[Query, GateExecutorResult] =

gateExecutor.arrow(gates, context)

override def inputAdaptor(

query: Query,

previousResult: RecommendationPipelineResult[Candidate, Result]

): Query =

query

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: GateExecutorResult

): RecommendationPipelineResult[Candidate, Result] =

previousPipelineResult.copy(gateResult = Some(executorResult))

}

def fetchQueryFeaturesStep(

queryFeatureHydrators: Seq[BaseQueryFeatureHydrator[Query, \_]],

stepIdentifier: PipelineStepIdentifier,

updater: ResultUpdater[

RecommendationPipelineResult[Candidate, Result],

QueryFeatureHydratorExecutor.Result

],

context: Executor.Context

): Step[Query, QueryFeatureHydratorExecutor.Result] =

new Step[Query, QueryFeatureHydratorExecutor.Result] {

override def identifier: PipelineStepIdentifier = stepIdentifier

override def executorArrow: Arrow[Query, QueryFeatureHydratorExecutor.Result] =

queryFeatureHydratorExecutor.arrow(

queryFeatureHydrators,

RecommendationPipelineConfig.stepsAsyncFeatureHydrationCanBeCompletedBy,

context)

override def inputAdaptor(

query: Query,

previousResult: RecommendationPipelineResult[Candidate, Result]

): Query = query

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: QueryFeatureHydratorExecutor.Result

): RecommendationPipelineResult[Candidate, Result] =

updater(previousPipelineResult, executorResult)

override def queryUpdater(

query: Query,

executorResult: QueryFeatureHydratorExecutor.Result

): Query =

query

.withFeatureMap(

query.features

.getOrElse(FeatureMap.empty) ++ executorResult.featureMap).asInstanceOf[Query]

}

def asyncFeaturesStep(

stepToHydrateFor: PipelineStepIdentifier,

context: Executor.Context

): Step[AsyncFeatureMap, AsyncFeatureMapExecutorResults] =

new Step[AsyncFeatureMap, AsyncFeatureMapExecutorResults] {

override def identifier: PipelineStepIdentifier =

RecommendationPipelineConfig.asyncFeaturesStep(stepToHydrateFor)

override def executorArrow: Arrow[AsyncFeatureMap, AsyncFeatureMapExecutorResults] =

asyncFeatureMapExecutor.arrow(

stepToHydrateFor,

identifier,

context

)

override def inputAdaptor(

query: Query,

previousResult: RecommendationPipelineResult[Candidate, Result]

): AsyncFeatureMap =

previousResult.mergedAsyncQueryFeatures

.getOrElse(

throw InvalidStepStateException(identifier, "MergedAsyncQueryFeatures")

)

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: AsyncFeatureMapExecutorResults

): RecommendationPipelineResult[Candidate, Result] =

previousPipelineResult.copy(

asyncFeatureHydrationResults = previousPipelineResult.asyncFeatureHydrationResults match {

case Some(existingResults) => Some(existingResults ++ executorResult)

case None => Some(executorResult)

})

override def queryUpdater(

query: Query,

executorResult: AsyncFeatureMapExecutorResults

): Query =

if (executorResult.featureMapsByStep

.getOrElse(stepToHydrateFor, FeatureMap.empty).isEmpty) {

query

} else {

query

.withFeatureMap(

query.features

.getOrElse(FeatureMap.empty) ++ executorResult.featureMapsByStep(

stepToHydrateFor)).asInstanceOf[Query]

}

}

def candidatePipelinesStep(

candidatePipelines: Seq[CandidatePipeline[Query]],

defaultFailOpenPolicy: FailOpenPolicy,

failOpenPolicies: Map[CandidatePipelineIdentifier, FailOpenPolicy],

qualityFactorObserverByPipeline: Map[ComponentIdentifier, QualityFactorObserver],

context: Executor.Context

): Step[CandidatePipeline.Inputs[Query], CandidatePipelineExecutorResult] =

new Step[CandidatePipeline.Inputs[Query], CandidatePipelineExecutorResult] {

override def identifier: PipelineStepIdentifier =

RecommendationPipelineConfig.candidatePipelinesStep

override def executorArrow: Arrow[CandidatePipeline.Inputs[

Query

], CandidatePipelineExecutorResult] =

candidatePipelineExecutor

.arrow(

candidatePipelines,

defaultFailOpenPolicy,

failOpenPolicies,

qualityFactorObserverByPipeline,

context

)

override def inputAdaptor(

query: Query,

previousResult: RecommendationPipelineResult[Candidate, Result]

): CandidatePipeline.Inputs[

Query

] = CandidatePipeline.Inputs(query, Seq.empty)

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: CandidatePipelineExecutorResult

): RecommendationPipelineResult[Candidate, Result] =

previousPipelineResult.copy(candidatePipelineResults = Some(executorResult))

override def queryUpdater(

query: Query,

executorResult: CandidatePipelineExecutorResult

): Query = {

val updatedFeatureMap = query.features

.getOrElse(FeatureMap.empty) ++ executorResult.queryFeatureMap

query

.withFeatureMap(updatedFeatureMap).asInstanceOf[Query]

}

}

def dependentCandidatePipelinesStep(

candidatePipelines: Seq[CandidatePipeline[Query]],

defaultFailOpenPolicy: FailOpenPolicy,

failOpenPolicies: Map[CandidatePipelineIdentifier, FailOpenPolicy],

qualityFactorObserverByPipeline: Map[ComponentIdentifier, QualityFactorObserver],

context: Executor.Context

): Step[CandidatePipeline.Inputs[Query], CandidatePipelineExecutorResult] =

new Step[CandidatePipeline.Inputs[Query], CandidatePipelineExecutorResult] {

override def identifier: PipelineStepIdentifier =

RecommendationPipelineConfig.dependentCandidatePipelinesStep

override def executorArrow: Arrow[CandidatePipeline.Inputs[

Query

], CandidatePipelineExecutorResult] =

candidatePipelineExecutor

.arrow(

candidatePipelines,

defaultFailOpenPolicy,

failOpenPolicies,

qualityFactorObserverByPipeline,

context

)

override def inputAdaptor(

query: Query,

previousResult: RecommendationPipelineResult[Candidate, Result]

): CandidatePipeline.Inputs[

Query

] = {

val previousCandidates = previousResult.candidatePipelineResults

.getOrElse {

throw InvalidStepStateException(identifier, "Candidates")

}.candidatePipelineResults.flatMap(\_.result.getOrElse(Seq.empty))

CandidatePipeline.Inputs(query, previousCandidates)

}

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: CandidatePipelineExecutorResult

): RecommendationPipelineResult[Candidate, Result] =

previousPipelineResult.copy(dependentCandidatePipelineResults = Some(executorResult))

override def queryUpdater(

query: Query,

executorResult: CandidatePipelineExecutorResult

): Query = {

val updatedFeatureMap = query.features

.getOrElse(FeatureMap.empty) ++ executorResult.queryFeatureMap

query

.withFeatureMap(updatedFeatureMap).asInstanceOf[Query]

}

}

abstract class FilterStep(

filters: Seq[Filter[Query, Candidate]],

context: Executor.Context,

override val identifier: PipelineStepIdentifier)

extends Step[

(Query, Seq[CandidateWithFeatures[Candidate]]),

FilterExecutorResult[Candidate]

] {

def itemCandidates(

previousResult: RecommendationPipelineResult[Candidate, Result]

): Seq[CandidateWithDetails]

override def executorArrow: Arrow[

(Query, Seq[CandidateWithFeatures[Candidate]]),

FilterExecutorResult[Candidate]

] =

filterExecutor.arrow(filters, context)

override def inputAdaptor(

query: Query,

previousResult: RecommendationPipelineResult[Candidate, Result]

): (Query, Seq[CandidateWithFeatures[Candidate]]) = {

val extractedItemCandidates = itemCandidates(previousResult).collect {

case itemCandidate: ItemCandidateWithDetails => itemCandidate

}

(query, extractedItemCandidates.asInstanceOf[Seq[CandidateWithFeatures[Candidate]]])

}

}

def postCandidatePipelinesSelectorStep(

selectors: Seq[Selector[Query]],

context: Executor.Context

): Step[SelectorExecutor.Inputs[Query], SelectorExecutorResult] =

new Step[SelectorExecutor.Inputs[Query], SelectorExecutorResult] {

override def identifier: PipelineStepIdentifier =

RecommendationPipelineConfig.postCandidatePipelinesSelectorsStep

override def executorArrow: Arrow[SelectorExecutor.Inputs[

Query

], SelectorExecutorResult] =

selectorExecutor.arrow(selectors, context)

override def inputAdaptor(

query: Query,

previousResult: RecommendationPipelineResult[Candidate, Result]

): SelectorExecutor.Inputs[Query] = {

val candidatePipelineResults = previousResult.candidatePipelineResults

.getOrElse {

throw InvalidStepStateException(identifier, "CandidatePipelineResults")

}.candidatePipelineResults.flatMap(\_.result.getOrElse(Seq.empty))

val dependentCandidatePipelineResults = previousResult.dependentCandidatePipelineResults

.getOrElse {

throw InvalidStepStateException(identifier, "DependentCandidatePipelineResults")

}.candidatePipelineResults.flatMap(\_.result.getOrElse(Seq.empty))

SelectorExecutor.Inputs(

query = query,

candidatesWithDetails = candidatePipelineResults ++ dependentCandidatePipelineResults

)

}

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: SelectorExecutorResult

): RecommendationPipelineResult[Candidate, Result] =

previousPipelineResult.copy(postCandidatePipelinesSelectorResults = Some(executorResult))

}

def postCandidatePipelinesFeatureHydrationStep(

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

context: Executor.Context

): Step[

CandidateFeatureHydratorExecutor.Inputs[Query, Candidate],

CandidateFeatureHydratorExecutorResult[Candidate]

] = new Step[

CandidateFeatureHydratorExecutor.Inputs[Query, Candidate],

CandidateFeatureHydratorExecutorResult[Candidate]

] {

override def identifier: PipelineStepIdentifier =

RecommendationPipelineConfig.postCandidatePipelinesFeatureHydrationStep

override def executorArrow: Arrow[

CandidateFeatureHydratorExecutor.Inputs[Query, Candidate],

CandidateFeatureHydratorExecutorResult[Candidate]

] =

candidateFeatureHydratorExecutor.arrow(hydrators, context)

override def inputAdaptor(

query: Query,

previousResult: RecommendationPipelineResult[Candidate, Result]

): CandidateFeatureHydratorExecutor.Inputs[Query, Candidate] = {

val selectedCandidatesResult =

previousResult.postCandidatePipelinesSelectorResults.getOrElse {

throw InvalidStepStateException(identifier, "PostCandidatePipelinesSelectorResults")

}.selectedCandidates

CandidateFeatureHydratorExecutor.Inputs(

query,

selectedCandidatesResult.asInstanceOf[Seq[CandidateWithFeatures[Candidate]]])

}

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: CandidateFeatureHydratorExecutorResult[Candidate]

): RecommendationPipelineResult[Candidate, Result] = previousPipelineResult.copy(

postCandidatePipelinesFeatureHydrationResults = Some(executorResult)

)

}

def globalFiltersStep(

filters: Seq[Filter[Query, Candidate]],

context: Executor.Context

): Step[(Query, Seq[CandidateWithFeatures[Candidate]]), FilterExecutorResult[Candidate]] =

new FilterStep(filters, context, RecommendationPipelineConfig.globalFiltersStep) {

override def itemCandidates(

previousResult: RecommendationPipelineResult[Candidate, Result]

): Seq[CandidateWithDetails] = {

val candidates = previousResult.postCandidatePipelinesSelectorResults

.getOrElse {

throw InvalidStepStateException(identifier, "PostCandidatePipelineSelectorResults")

}.selectedCandidates.collect {

case itemCandidate: ItemCandidateWithDetails => itemCandidate

}

val featureMaps = previousResult.postCandidatePipelinesFeatureHydrationResults

.getOrElse {

throw InvalidStepStateException(

identifier,

"PostCandidatePipelineFeatureHydrationResults")

}.results.map(\_.features)

// If no hydrators were run, this list would be empty. Otherwise, order and cardinality is

// always ensured to match.

if (featureMaps.isEmpty) {

candidates

} else {

candidates.zip(featureMaps).map {

case (candidate, featureMap) =>

candidate.copy(features = candidate.features ++ featureMap)

}

}

}

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: FilterExecutorResult[Candidate]

): RecommendationPipelineResult[Candidate, Result] = previousPipelineResult.copy(

globalFilterResults = Some(executorResult)

)

}

def scoringPipelinesStep(

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

context: Executor.Context,

defaultFailOpenPolicy: FailOpenPolicy,

failOpenPolicies: Map[ScoringPipelineIdentifier, FailOpenPolicy],

qualityFactorObserverByPipeline: Map[ComponentIdentifier, QualityFactorObserver]

): Step[ScoringPipelineExecutor.Inputs[Query], ScoringPipelineExecutorResult[

Candidate

]] =

new Step[ScoringPipelineExecutor.Inputs[Query], ScoringPipelineExecutorResult[

Candidate

]] {

override def identifier: PipelineStepIdentifier =

RecommendationPipelineConfig.scoringPipelinesStep

override def executorArrow: Arrow[

ScoringPipelineExecutor.Inputs[Query],

ScoringPipelineExecutorResult[Candidate]

] = scoringPipelineExecutor.arrow(

scoringPipelines,

context,

defaultFailOpenPolicy,

failOpenPolicies,

qualityFactorObserverByPipeline

)

override def inputAdaptor(

query: Query,

previousResult: RecommendationPipelineResult[Candidate, Result]

): ScoringPipelineExecutor.Inputs[Query] = {

val selectedCandidates =

previousResult.postCandidatePipelinesSelectorResults.getOrElse {

throw InvalidStepStateException(identifier, "PostCandidatePipelinesSelectorResults")

}.selectedCandidates

val itemCandidates = selectedCandidates.collect {

case itemCandidate: ItemCandidateWithDetails => itemCandidate

}

val featureMaps = previousResult.postCandidatePipelinesFeatureHydrationResults

.getOrElse {

throw InvalidStepStateException(

identifier,

"PostCandidatePipelineFeatureHydrationResults")

}.results.map(\_.features)

// If no hydrators were run, this list would be empty. Otherwise, order and cardinality is

// always ensured to match.

val updatedCandidates = if (featureMaps.isEmpty) {

itemCandidates

} else {

itemCandidates.zip(featureMaps).map {

case (candidate, featureMap) =>

candidate.copy(features = candidate.features ++ featureMap)

}

}

// Filter the original list of candidates to keep only the ones that were kept from

// filtering

val filterResults: Set[Candidate] = previousResult.globalFilterResults

.getOrElse {

throw InvalidStepStateException(identifier, "FilterResults")

}.result.toSet

val filteredItemCandidates = updatedCandidates.filter { itemCandidate =>

filterResults.contains(itemCandidate.candidate.asInstanceOf[Candidate])

}

ScoringPipelineExecutor.Inputs(

query,

filteredItemCandidates

)

}

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: ScoringPipelineExecutorResult[Candidate]

): RecommendationPipelineResult[Candidate, Result] = previousPipelineResult

.copy(scoringPipelineResults = Some(executorResult))

}

def resultSelectorsStep(

selectors: Seq[Selector[Query]],

context: Executor.Context

): Step[SelectorExecutor.Inputs[Query], SelectorExecutorResult] =

new Step[SelectorExecutor.Inputs[Query], SelectorExecutorResult] {

override def identifier: PipelineStepIdentifier =

RecommendationPipelineConfig.resultSelectorsStep

override def executorArrow: Arrow[SelectorExecutor.Inputs[

Query

], SelectorExecutorResult] =

selectorExecutor.arrow(selectors, context)

override def inputAdaptor(

query: Query,

previousResult: RecommendationPipelineResult[Candidate, Result]

): SelectorExecutor.Inputs[Query] = {

/\*\*

\* See [[ScoringPipelineExecutor]], scoringPipelineResults contains the fully re-merged

\* and updated FeatureMap so there's no need to do any recomposition. Scoring Pipeline Results

\* has only candidates that were kept in previous filtering, with their final merged feature

\* map.

\*/

val scorerResults = previousResult.scoringPipelineResults.getOrElse {

throw InvalidStepStateException(identifier, "Scores")

}

SelectorExecutor.Inputs(

query = query,

candidatesWithDetails = scorerResults.result

)

}

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: SelectorExecutorResult

): RecommendationPipelineResult[Candidate, Result] =

previousPipelineResult.copy(resultSelectorResults = Some(executorResult))

}

def postSelectionFiltersStep(

filters: Seq[Filter[Query, Candidate]],

context: Executor.Context

): Step[(Query, Seq[CandidateWithFeatures[Candidate]]), FilterExecutorResult[Candidate]] =

new FilterStep(filters, context, RecommendationPipelineConfig.postSelectionFiltersStep) {

override def itemCandidates(

previousResult: RecommendationPipelineResult[Candidate, Result]

): Seq[CandidateWithDetails] = {

previousResult.resultSelectorResults.getOrElse {

throw InvalidStepStateException(identifier, "Candidates")

}.selectedCandidates

}

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: FilterExecutorResult[Candidate]

): RecommendationPipelineResult[Candidate, Result] = {

previousPipelineResult.copy(postSelectionFilterResults = Some(executorResult))

}

}

def decoratorStep(

decorator: Option[CandidateDecorator[Query, Candidate]],

context: Executor.Context

): Step[(Query, Seq[CandidateWithFeatures[Candidate]]), CandidateDecoratorExecutorResult] =

new Step[(Query, Seq[CandidateWithFeatures[Candidate]]), CandidateDecoratorExecutorResult] {

override def identifier: PipelineStepIdentifier = RecommendationPipelineConfig.decoratorStep

override lazy val executorArrow: Arrow[

(Query, Seq[CandidateWithFeatures[Candidate]]),

CandidateDecoratorExecutorResult

] =

candidateDecoratorExecutor.arrow(decorator, context)

override def inputAdaptor(

query: Query,

previousResult: RecommendationPipelineResult[Candidate, Result]

): (Query, Seq[CandidateWithFeatures[Candidate]]) = {

val selectorResults = previousResult.resultSelectorResults

.getOrElse {

throw InvalidStepStateException(identifier, "SelectorResults")

}.selectedCandidates

.collect { case candidate: ItemCandidateWithDetails => candidate }

val filterResults = previousResult.postSelectionFilterResults

.getOrElse {

throw InvalidStepStateException(identifier, "PostSelectionFilterResults")

}.result.toSet

val itemCandidateWithDetailsPostFiltering =

selectorResults

.filter(candidateWithDetails =>

filterResults.contains(

candidateWithDetails.candidate

.asInstanceOf[Candidate]))

.asInstanceOf[Seq[CandidateWithFeatures[Candidate]]]

(query, itemCandidateWithDetailsPostFiltering)

}

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: CandidateDecoratorExecutorResult

): RecommendationPipelineResult[Candidate, Result] =

previousPipelineResult.copy(

candidateDecoratorResult = Some(executorResult)

)

}

def domainMarshallingStep(

domainMarshaller: DomainMarshaller[Query, DomainResultType],

context: Executor.Context

): Step[DomainMarshallerExecutor.Inputs[Query], DomainMarshallerExecutor.Result[

DomainResultType

]] =

new Step[DomainMarshallerExecutor.Inputs[Query], DomainMarshallerExecutor.Result[

DomainResultType

]] {

override def identifier: PipelineStepIdentifier =

RecommendationPipelineConfig.domainMarshallerStep

override def executorArrow: Arrow[

DomainMarshallerExecutor.Inputs[Query],

DomainMarshallerExecutor.Result[DomainResultType]

] =

domainMarshallerExecutor.arrow(domainMarshaller, context)

override def inputAdaptor(

query: Query,

previousResult: RecommendationPipelineResult[Candidate, Result]

): DomainMarshallerExecutor.Inputs[Query] = {

val selectorResults = previousResult.resultSelectorResults.getOrElse {

throw InvalidStepStateException(identifier, "SelectorResults")

}

val filterResults = previousResult.postSelectionFilterResults

.getOrElse {

throw InvalidStepStateException(identifier, "PostSelectionFilterResults")

}.result.toSet

val filteredResults = selectorResults.selectedCandidates.collect {

case candidate: ItemCandidateWithDetails

if filterResults.contains(candidate.candidate.asInstanceOf[Candidate]) =>

candidate

}

val decoratorResults = previousResult.candidateDecoratorResult

.getOrElse(throw InvalidStepStateException(identifier, "DecoratorStep")).result.map {

decoration =>

decoration.candidate -> decoration.presentation

}.toMap

val finalResults = filteredResults.map { itemWithDetails =>

decoratorResults.get(itemWithDetails.candidate) match {

case Some(presentation: ItemPresentation) =>

if (itemWithDetails.presentation.isDefined) {

throw PipelineFailure(

category = MisconfiguredDecorator,

reason = "Item Candidate already decorated",

componentStack = Some(context.componentStack))

} else {

itemWithDetails.copy(presentation = Some(presentation))

}

case Some(\_) =>

throw PipelineFailure(

category = MisconfiguredDecorator,

reason = "Item Candidate got back a non ItemPresentation from decorator",

componentStack = Some(context.componentStack))

case None => itemWithDetails

}

}

DomainMarshallerExecutor.Inputs(

query = query,

candidatesWithDetails = finalResults

)

}

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: DomainMarshallerExecutor.Result[DomainResultType]

): RecommendationPipelineResult[Candidate, Result] = previousPipelineResult.copy(

domainMarshallerResults = Some(executorResult)

)

}

def resultSideEffectsStep(

sideEffects: Seq[PipelineResultSideEffect[Query, DomainResultType]],

context: Executor.Context

): Step[

PipelineResultSideEffect.Inputs[Query, DomainResultType],

PipelineResultSideEffectExecutor.Result

] = new Step[

PipelineResultSideEffect.Inputs[Query, DomainResultType],

PipelineResultSideEffectExecutor.Result

] {

override def identifier: PipelineStepIdentifier =

RecommendationPipelineConfig.resultSideEffectsStep

override def executorArrow: Arrow[

PipelineResultSideEffect.Inputs[Query, DomainResultType],

PipelineResultSideEffectExecutor.Result

] = pipelineResultSideEffectExecutor.arrow(sideEffects, context)

override def inputAdaptor(

query: Query,

previousResult: RecommendationPipelineResult[Candidate, Result]

): PipelineResultSideEffect.Inputs[Query, DomainResultType] = {

// Re-apply decorations to the selected results

val resultSelectorResults = {

val decoratorResults = previousResult.candidateDecoratorResult

.getOrElse(throw InvalidStepStateException(identifier, "DecoratorStep")).result.map {

decoration =>

decoration.candidate -> decoration.presentation

}.toMap

val previousSelectorResults = previousResult.resultSelectorResults.getOrElse {

throw InvalidStepStateException(identifier, "SelectorResults")

}

val filterResults = previousResult.postSelectionFilterResults

.getOrElse {

throw InvalidStepStateException(identifier, "PostSelectionFilterResults")

}.result.toSet

val filteredSelectorResults = previousSelectorResults.selectedCandidates.collect {

case candidate: ItemCandidateWithDetails

if filterResults.contains(candidate.candidate.asInstanceOf[Candidate]) =>

candidate

}

val decoratedSelectedResults = filteredSelectorResults.map {

case itemWithDetails: ItemCandidateWithDetails =>

decoratorResults.get(itemWithDetails.candidate) match {

case Some(presentation: ItemPresentation) =>

if (itemWithDetails.presentation.isDefined) {

throw PipelineFailure(

category = MisconfiguredDecorator,

reason = "Item Candidate already decorated",

componentStack = Some(context.componentStack))

} else {

itemWithDetails.copy(presentation = Some(presentation))

}

case Some(\_) =>

throw PipelineFailure(

category = MisconfiguredDecorator,

reason = "Item Candidate got back a non ItemPresentation from decorator",

componentStack = Some(context.componentStack))

case None => itemWithDetails

}

case item =>

// This branch should be impossible to hit since we do a .collect on ItemCandidateWithDetails

// as part of executing the candidate pipelines.

throw PipelineFailure(

category = IllegalStateFailure,

reason =

s"Only ItemCandidateWithDetails expected in pipeline, found: ${item.toString}",

componentStack = Some(context.componentStack)

)

}

previousSelectorResults.copy(selectedCandidates = decoratedSelectedResults)

}

val domainMarshallerResults = previousResult.domainMarshallerResults.getOrElse {

throw InvalidStepStateException(identifier, "DomainMarshallerResults")

}

PipelineResultSideEffect.Inputs[Query, DomainResultType](

query = query,

selectedCandidates = resultSelectorResults.selectedCandidates,

remainingCandidates = resultSelectorResults.remainingCandidates,

droppedCandidates = resultSelectorResults.droppedCandidates,

response = domainMarshallerResults.result.asInstanceOf[DomainResultType]

)

}

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: PipelineResultSideEffectExecutor.Result

): RecommendationPipelineResult[Candidate, Result] =

previousPipelineResult.copy(resultSideEffectResults = Some(executorResult))

}

def transportMarshallingStep(

transportMarshaller: TransportMarshaller[DomainResultType, Result],

context: Executor.Context

): Step[

TransportMarshallerExecutor.Inputs[DomainResultType],

TransportMarshallerExecutor.Result[Result]

] = new Step[TransportMarshallerExecutor.Inputs[

DomainResultType

], TransportMarshallerExecutor.Result[Result]] {

override def identifier: PipelineStepIdentifier =

RecommendationPipelineConfig.transportMarshallerStep

override def executorArrow: Arrow[TransportMarshallerExecutor.Inputs[

DomainResultType

], TransportMarshallerExecutor.Result[Result]] =

transportMarshallerExecutor.arrow(transportMarshaller, context)

override def inputAdaptor(

query: Query,

previousResult: RecommendationPipelineResult[Candidate, Result]

): TransportMarshallerExecutor.Inputs[DomainResultType] = {

val domainMarshallingResults = previousResult.domainMarshallerResults.getOrElse {

throw InvalidStepStateException(identifier, "DomainMarshallerResults")

}

// Since the PipelineResult just uses HasMarshalling

val domainResult = domainMarshallingResults.result.asInstanceOf[DomainResultType]

TransportMarshallerExecutor.Inputs(domainResult)

}

override def resultUpdater(

previousPipelineResult: RecommendationPipelineResult[Candidate, Result],

executorResult: TransportMarshallerExecutor.Result[Result]

): RecommendationPipelineResult[Candidate, Result] = previousPipelineResult.copy(

transportMarshallerResults = Some(executorResult),

result = Some(executorResult.result)

)

}

def build(

parentComponentIdentifierStack: ComponentIdentifierStack,

config: RecommendationPipelineConfig[

Query,

Candidate,

DomainResultType,

Result

]

): RecommendationPipeline[Query, Candidate, Result] = {

val pipelineIdentifier = config.identifier

val context = Executor.Context(

PipelineFailureClassifier(

config.failureClassifier.orElse(StoppedGateException.classifier(ProductDisabled))),

parentComponentIdentifierStack.push(pipelineIdentifier)

)

val decorator = config.decorator.map(decorator =>

CandidateDecorator.copyWithUpdatedIdentifier(decorator, pipelineIdentifier))

val qualityFactorStatus: QualityFactorStatus =

QualityFactorStatus.build(config.qualityFactorConfigs)

val qualityFactorObserverByPipeline =

qualityFactorStatus.qualityFactorByPipeline.mapValues { qualityFactor =>

qualityFactor.buildObserver()

}

buildGaugesForQualityFactor(pipelineIdentifier, qualityFactorStatus, statsReceiver)

val candidatePipelines: Seq[CandidatePipeline[Query]] = config.candidatePipelines.map {

pipelineConfig: CandidatePipelineConfig[Query, \_, \_, \_] =>

pipelineConfig.build(context.componentStack, candidatePipelineBuilderFactory)

}

val dependentCandidatePipelines: Seq[CandidatePipeline[Query]] =

config.dependentCandidatePipelines.map {

pipelineConfig: DependentCandidatePipelineConfig[Query, \_, \_, \_] =>

pipelineConfig.build(context.componentStack, candidatePipelineBuilderFactory)

}

val scoringPipelines: Seq[ScoringPipeline[Query, Candidate]] = config.scoringPipelines.map {

pipelineConfig: ScoringPipelineConfig[Query, Candidate] =>

pipelineConfig.build(context.componentStack, scoringPipelineBuilderFactory)

}

val builtSteps = Seq(

qualityFactorStep(qualityFactorStatus),

gatesStep(config.gates, context),

fetchQueryFeaturesStep(

config.fetchQueryFeatures,

RecommendationPipelineConfig.fetchQueryFeaturesStep,

(previousPipelineResult, executorResult) =>

previousPipelineResult.copy(queryFeatures = Some(executorResult)),

context

),

fetchQueryFeaturesStep(

config.fetchQueryFeaturesPhase2,

RecommendationPipelineConfig.fetchQueryFeaturesPhase2Step,

(previousPipelineResult, executorResult) =>

previousPipelineResult.copy(

queryFeaturesPhase2 = Some(executorResult),

mergedAsyncQueryFeatures = Some(

previousPipelineResult.queryFeatures

.getOrElse(throw InvalidStepStateException(

RecommendationPipelineConfig.fetchQueryFeaturesPhase2Step,

"QueryFeatures"))

.asyncFeatureMap ++ executorResult.asyncFeatureMap)

),

context

),

asyncFeaturesStep(RecommendationPipelineConfig.candidatePipelinesStep, context),

candidatePipelinesStep(

candidatePipelines,

config.defaultFailOpenPolicy,

config.candidatePipelineFailOpenPolicies,

qualityFactorObserverByPipeline,

context),

asyncFeaturesStep(RecommendationPipelineConfig.dependentCandidatePipelinesStep, context),

dependentCandidatePipelinesStep(

dependentCandidatePipelines,

config.defaultFailOpenPolicy,

config.candidatePipelineFailOpenPolicies,

qualityFactorObserverByPipeline,

context),

asyncFeaturesStep(RecommendationPipelineConfig.postCandidatePipelinesSelectorsStep, context),

postCandidatePipelinesSelectorStep(config.postCandidatePipelinesSelectors, context),

asyncFeaturesStep(

RecommendationPipelineConfig.postCandidatePipelinesFeatureHydrationStep,

context),

postCandidatePipelinesFeatureHydrationStep(

config.postCandidatePipelinesFeatureHydration,

context),

asyncFeaturesStep(RecommendationPipelineConfig.globalFiltersStep, context),

globalFiltersStep(config.globalFilters, context),

asyncFeaturesStep(RecommendationPipelineConfig.scoringPipelinesStep, context),

scoringPipelinesStep(

scoringPipelines,

context,

config.defaultFailOpenPolicy,

config.scoringPipelineFailOpenPolicies,

qualityFactorObserverByPipeline

),

asyncFeaturesStep(RecommendationPipelineConfig.resultSelectorsStep, context),

resultSelectorsStep(config.resultSelectors, context),

asyncFeaturesStep(RecommendationPipelineConfig.postSelectionFiltersStep, context),

postSelectionFiltersStep(config.postSelectionFilters, context),

asyncFeaturesStep(RecommendationPipelineConfig.decoratorStep, context),

decoratorStep(decorator, context),

domainMarshallingStep(config.domainMarshaller, context),

asyncFeaturesStep(RecommendationPipelineConfig.resultSideEffectsStep, context),

resultSideEffectsStep(config.resultSideEffects, context),

transportMarshallingStep(config.transportMarshaller, context)

)

val finalArrow = buildCombinedArrowFromSteps(

steps = builtSteps,

context = context,

initialEmptyResult = RecommendationPipelineResult.empty,

stepsInOrderFromConfig = RecommendationPipelineConfig.stepsInOrder

)

val configFromBuilder = config

new RecommendationPipeline[Query, Candidate, Result] {

override private[core] val config: RecommendationPipelineConfig[

Query,

Candidate,

\_,

Result

] =

configFromBuilder

override val arrow: Arrow[Query, RecommendationPipelineResult[Candidate, Result]] =

finalArrow

override val identifier: RecommendationPipelineIdentifier = pipelineIdentifier

override val alerts: Seq[Alert] = config.alerts

override val children: Seq[Component] =

config.gates ++

config.fetchQueryFeatures ++

candidatePipelines ++

dependentCandidatePipelines ++

config.postCandidatePipelinesFeatureHydration ++

config.globalFilters ++

scoringPipelines ++

config.postSelectionFilters ++

config.resultSideEffects ++

decorator.toSeq ++

Seq(config.domainMarshaller, config.transportMarshaller)

}

}

}