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

import com.twitter.product\_mixer.core.functional\_component.candidate\_source.BaseCandidateSource

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.BaseGate

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

import com.twitter.product\_mixer.core.functional\_component.scorer.Scorer

import com.twitter.product\_mixer.core.functional\_component.transformer.CandidateFeatureTransformer

import com.twitter.product\_mixer.core.functional\_component.transformer.CandidatePipelineQueryTransformer

import com.twitter.product\_mixer.core.functional\_component.transformer.CandidatePipelineResultsTransformer

import com.twitter.product\_mixer.core.functional\_component.transformer.\_

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.ComponentIdentifierStack

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

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

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

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

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

import com.twitter.timelines.configapi.FSParam

import com.twitter.timelines.configapi.decider.DeciderParam

sealed trait BaseCandidatePipelineConfig[

-Query <: PipelineQuery,

CandidateSourceQuery,

CandidateSourceResult,

Result <: UniversalNoun[Any]]

extends PipelineConfig {

val identifier: CandidatePipelineIdentifier

/\*\*

\* A candidate pipeline can fetch query-level features for use within the candidate source. It's

\* generally recommended to set a hydrator in the parent recos or mixer pipeline if multiple

\* candidate pipelines share the same feature but if a specific query feature hydrator is used

\* by one pipeline and you don't want to block the others, you could explicitly set it here.

\* If a feature is hydrated both in the parent pipeline or here, this one takes priority.

\*/

def queryFeatureHydration: Seq[BaseQueryFeatureHydrator[Query, \_]] = Seq.empty

/\*\*

\* For query-level features that are dependent on query-level features from [[queryFeatureHydration]]

\*/

def queryFeatureHydrationPhase2: Seq[BaseQueryFeatureHydrator[Query, \_]] = Seq.empty

/\*\*

\* When these Params are defined, they will automatically be added as Gates in the pipeline

\* by the CandidatePipelineBuilder

\*

\* The enabled decider param can to be used to quickly disable a Candidate Pipeline via Decider

\*/

val enabledDeciderParam: Option[DeciderParam[Boolean]] = None

/\*\*

\* This supported client feature switch param can be used with a Feature Switch to control the

\* rollout of a new Candidate Pipeline from dogfood to experiment to production

\*/

val supportedClientParam: Option[FSParam[Boolean]] = None

/\*\* [[Gate]]s that are applied sequentially, the pipeline will only run if all the Gates are open \*/

def gates: Seq[BaseGate[Query]] = Seq.empty

/\*\*

\* A pair of transforms to adapt the underlying candidate source to the pipeline's query and result types

\* Complex use cases such as those that need access to features should construct their own transformer, but

\* for simple use cases, you can pass in an anonymous function.

\* @example

\* {{{ override val queryTransformer: CandidatePipelineQueryTransformer[Query, CandidateSourceQuery] = { query =>

\* query.toExampleThrift

\* }

\* }}}

\*/

def queryTransformer: BaseCandidatePipelineQueryTransformer[

Query,

CandidateSourceQuery

]

/\*\* Source for Candidates for this Pipeline \*/

def candidateSource: BaseCandidateSource[CandidateSourceQuery, CandidateSourceResult]

/\*\*

\* [[CandidateFeatureTransformer]] allow you to define [[com.twitter.product\_mixer.core.feature.Feature]] extraction logic from your [[CandidateSource]] results.

\* If your candidate sources return [[com.twitter.product\_mixer.core.feature.Feature]]s alongside the candidate that might be useful later on,

\* add transformers for constructing feature maps.

\*

\* @note If multiple transformers extract the same feature, the last one takes priority and is kept.

\*/

def featuresFromCandidateSourceTransformers: Seq[

CandidateFeatureTransformer[CandidateSourceResult]

] = Seq.empty

/\*\*

\* a result Transformer may throw PipelineFailure for candidates that are malformed and

\* should be removed. This should be exceptional behavior, and not a replacement for adding a Filter.

\* Complex use cases such as those that need access to features should construct their own transformer, but

\* for simple use cases, you can pass in an anonymous function.

\* @example

\* {{{ override val queryTransformer: CandidatePipelineResultsTransformer[CandidateSourceResult, Result] = { sourceResult =>

\* ExampleCandidate(sourceResult.id)

\* }

\* }}}

\*

\*/

val resultTransformer: CandidatePipelineResultsTransformer[CandidateSourceResult, Result]

/\*\*

\* Before filters are run, you can fetch features for each candidate.

\*

\* Uses Stitch, so you're encouraged to use a working Stitch Adaptor to batch between candidates.

\*

\* The existing features (from the candidate source) are passed in as an input. You are not expected

\* to put them into the resulting feature map yourself - they will be merged for you by the platform.

\*

\* This API is likely to change when Product Mixer does managed feature hydration

\*/

val preFilterFeatureHydrationPhase1: Seq[BaseCandidateFeatureHydrator[Query, Result, \_]] =

Seq.empty

/\*\*

\* A second phase of feature hydration that can be run before filtering and after the first phase

\* of [[preFilterFeatureHydrationPhase1]]. You are not expected to put them into the resulting

\* feature map yourself - they will be merged for you by the platform.

\*/

val preFilterFeatureHydrationPhase2: Seq[BaseCandidateFeatureHydrator[Query, Result, \_]] =

Seq.empty

/\*\* A list of filters to apply. Filters will be applied in sequential order. \*/

def filters: Seq[Filter[Query, Result]] = Seq.empty

/\*\*

\* After filters are run, you can fetch features for each candidate.

\*

\* Uses Stitch, so you're encouraged to use a working Stitch Adaptor to batch between candidates.

\*

\* The existing features (from the candidate source) & pre-filtering are passed in as an input.

\* You are not expected to put them into the resulting feature map yourself -

\* they will be merged for you by the platform.

\*

\* This API is likely to change when Product Mixer does managed feature hydration

\*/

val postFilterFeatureHydration: Seq[BaseCandidateFeatureHydrator[Query, Result, \_]] = Seq.empty

/\*\*

\* Decorators allow for adding Presentations to candidates. While the Presentation can contain any

\* arbitrary data, Decorators are often used to add a UrtItemPresentation for URT item support, or

\* a UrtModulePresentation for grouping the candidates in a URT module.

\*/

val decorator: Option[CandidateDecorator[Query, Result]] = None

/\*\*

\* A candidate pipeline can define a partial function to rescue failures here. They will be treated as failures

\* from a monitoring standpoint, and cancellation exceptions will always be propagated (they cannot be caught here).

\*/

def failureClassifier: PartialFunction[Throwable, PipelineFailure] = PartialFunction.empty

/\*\*

\* Scorers for candidates. Scorers are executed in parallel. Order does not matter.

\*/

def scorers: Seq[Scorer[Query, Result]] = Seq.empty

/\*\*

\* Alerts can be used to indicate the pipeline's service level objectives. Alerts and

\* dashboards will be automatically created based on this information.

\*/

val alerts: Seq[Alert] = Seq.empty

/\*\*

\* This method is used by the product mixer framework to build the pipeline.

\*/

private[core] final def build(

parentComponentIdentifierStack: ComponentIdentifierStack,

factory: CandidatePipelineBuilderFactory

): CandidatePipeline[Query] = {

factory.get.build(parentComponentIdentifierStack, this)

}

}

trait CandidatePipelineConfig[

-Query <: PipelineQuery,

CandidateSourceQuery,

CandidateSourceResult,

Result <: UniversalNoun[Any]]

extends BaseCandidatePipelineConfig[

Query,

CandidateSourceQuery,

CandidateSourceResult,

Result

] {

override val gates: Seq[Gate[Query]] = Seq.empty

override val queryTransformer: CandidatePipelineQueryTransformer[

Query,

CandidateSourceQuery

]

}

trait DependentCandidatePipelineConfig[

-Query <: PipelineQuery,

CandidateSourceQuery,

CandidateSourceResult,

Result <: UniversalNoun[Any]]

extends BaseCandidatePipelineConfig[

Query,

CandidateSourceQuery,

CandidateSourceResult,

Result

]

/\*\*

\* Contains [[PipelineStepIdentifier]]s for the Steps that are available for all [[BaseCandidatePipelineConfig]]s

\*/

object CandidatePipelineConfig extends PipelineConfigCompanion {

val gatesStep: PipelineStepIdentifier = PipelineStepIdentifier("Gates")

val fetchQueryFeaturesStep: PipelineStepIdentifier = PipelineStepIdentifier("FetchQueryFeatures")

val fetchQueryFeaturesPhase2Step: PipelineStepIdentifier = PipelineStepIdentifier(

"FetchQueryFeaturesPhase2")

val candidateSourceStep: PipelineStepIdentifier = PipelineStepIdentifier("CandidateSource")

val preFilterFeatureHydrationPhase1Step: PipelineStepIdentifier =

PipelineStepIdentifier("PreFilterFeatureHydration")

val preFilterFeatureHydrationPhase2Step: PipelineStepIdentifier =

PipelineStepIdentifier("PreFilterFeatureHydrationPhase2")

val filtersStep: PipelineStepIdentifier = PipelineStepIdentifier("Filters")

val postFilterFeatureHydrationStep: PipelineStepIdentifier =

PipelineStepIdentifier("PostFilterFeatureHydration")

val scorersStep: PipelineStepIdentifier = PipelineStepIdentifier("Scorer")

val decoratorStep: PipelineStepIdentifier = PipelineStepIdentifier("Decorator")

val resultStep: PipelineStepIdentifier = PipelineStepIdentifier("Result")

/\*\* All the steps which are executed by a [[CandidatePipeline]] in the order in which they are run \*/

override val stepsInOrder: Seq[PipelineStepIdentifier] = Seq(

gatesStep,

fetchQueryFeaturesStep,

fetchQueryFeaturesPhase2Step,

asyncFeaturesStep(candidateSourceStep),

candidateSourceStep,

asyncFeaturesStep(preFilterFeatureHydrationPhase1Step),

preFilterFeatureHydrationPhase1Step,

asyncFeaturesStep(preFilterFeatureHydrationPhase2Step),

preFilterFeatureHydrationPhase2Step,

asyncFeaturesStep(filtersStep),

filtersStep,

asyncFeaturesStep(postFilterFeatureHydrationStep),

postFilterFeatureHydrationStep,

asyncFeaturesStep(scorersStep),

scorersStep,

asyncFeaturesStep(decoratorStep),

decoratorStep,

resultStep

)

override val stepsAsyncFeatureHydrationCanBeCompletedBy: Set[PipelineStepIdentifier] = Set(

candidateSourceStep,

preFilterFeatureHydrationPhase1Step,

preFilterFeatureHydrationPhase2Step,

filtersStep,

postFilterFeatureHydrationStep,

scorersStep,

decoratorStep

)

}