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

import com.twitter.product\_mixer.component\_library.selector.InsertAppendResults

import com.twitter.product\_mixer.core.functional\_component.common.AllPipelines

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

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

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

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

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

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

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

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.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

/\*\*

\* This is the configuration necessary to generate a Scoring Pipeline. Product code should create a

\* ScoringPipelineConfig, and then use a ScoringPipelineBuilder to get the final ScoringPipeline which can

\* process requests.

\*

\* @tparam Query - The domain model for the query or request

\* @tparam Candidate the domain model for the candidate being scored

\*/

trait ScoringPipelineConfig[-Query <: PipelineQuery, Candidate <: UniversalNoun[Any]]

extends PipelineConfig {

override val identifier: ScoringPipelineIdentifier

/\*\*

\* 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

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

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

/\*\*

\* Logic for selecting which candidates to score. Note, this doesn't drop the candidates from

\* the final result, just whether to score it in this pipeline or not.

\*/

def selectors: Seq[Selector[Query]]

/\*\*

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

\* The existing features from previous hydrations are passed in as inputs. You are not expected to

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

\*/

def preScoringFeatureHydrationPhase1: Seq[BaseCandidateFeatureHydrator[Query, Candidate, \_]] =

Seq.empty

/\*\*

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

\* of pre-scoring feature hydration. You are not expected to put them into the resulting

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

\*/

def preScoringFeatureHydrationPhase2: Seq[BaseCandidateFeatureHydrator[Query, Candidate, \_]] =

Seq.empty

/\*\*

\* Ranker Function for candidates. Scorers are executed in parallel.

\* Note: Order does not matter, this could be a Set if Set was covariant over it's type.

\*/

def scorers: Seq[Scorer[Query, Candidate]]

/\*\*

\* A 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

/\*\*

\* 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,

builder: ScoringPipelineBuilderFactory

): ScoringPipeline[Query, Candidate] =

builder.get.build(parentComponentIdentifierStack, this)

}

object ScoringPipelineConfig extends PipelineConfigCompanion {

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

scorer: Scorer[Query, Candidate]

): ScoringPipelineConfig[Query, Candidate] = new ScoringPipelineConfig[Query, Candidate] {

override val identifier: ScoringPipelineIdentifier = ScoringPipelineIdentifier(

s"ScoreAll${scorer.identifier.name}")

override val selectors: Seq[Selector[Query]] = Seq(InsertAppendResults(AllPipelines))

override val scorers: Seq[Scorer[Query, Candidate]] = Seq(scorer)

}

val gatesStep: PipelineStepIdentifier = PipelineStepIdentifier("Gates")

val selectorsStep: PipelineStepIdentifier = PipelineStepIdentifier("Selectors")

val preScoringFeatureHydrationPhase1Step: PipelineStepIdentifier =

PipelineStepIdentifier("PreScoringFeatureHydrationPhase1")

val preScoringFeatureHydrationPhase2Step: PipelineStepIdentifier =

PipelineStepIdentifier("PreScoringFeatureHydrationPhase2")

val scorersStep: PipelineStepIdentifier = PipelineStepIdentifier("Scorers")

val resultStep: PipelineStepIdentifier = PipelineStepIdentifier("Result")

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

override val stepsInOrder: Seq[PipelineStepIdentifier] = Seq(

gatesStep,

selectorsStep,

preScoringFeatureHydrationPhase1Step,

preScoringFeatureHydrationPhase2Step,

scorersStep,

resultStep

)

}