package com.twitter.follow\_recommendations.common.base

import com.twitter.finagle.stats.StatsReceiver

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

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

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

import com.twitter.product\_mixer.core.pipeline.recommendation.RecommendationPipelineResult

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

import com.twitter.stitch.Stitch

/\*\*

\* configs for results generated from the recommendation flow

\*

\* @param desiredCandidateCount num of desired candidates to return

\* @param batchForCandidatesCheck batch size for candidates check

\*/

case class RecommendationResultsConfig(desiredCandidateCount: Int, batchForCandidatesCheck: Int)

trait BaseRecommendationFlow[Target, Candidate <: UniversalNoun[Long]] {

val identifier = RecommendationPipelineIdentifier("RecommendationFlow")

def process(

pipelineRequest: Target

): Stitch[RecommendationPipelineResult[Candidate, Seq[Candidate]]]

def mapKey[Target2](fn: Target2 => Target): BaseRecommendationFlow[Target2, Candidate] = {

val original = this

new BaseRecommendationFlow[Target2, Candidate] {

override def process(

pipelineRequest: Target2

): Stitch[RecommendationPipelineResult[Candidate, Seq[Candidate]]] =

original.process(fn(pipelineRequest))

}

}

}

/\*\*

\* Defines a typical recommendation flow to fetch, filter, rank and transform candidates.

\*

\* 1. targetEligibility: determine the eligibility of target request

\* 2. candidateSources: fetch candidates from candidate sources based on target type

\* 3. preRankerCandidateFilter: light filtering of candidates

\* 4. ranker: ranking of candidates (could be composed of multiple stages, light ranking, heavy ranking and etc)

\* 5. postRankerTransform: deduping, grouping, rule based promotion / demotions and etc

\* 6. validateCandidates: heavy filters to determine the eligibility of the candidates.

\* will only be applied to candidates that we expect to return.

\* 7. transformResults: transform the individual candidates into desired format (e.g. hydrate social proof)

\*

\* Note that the actual implementations may not need to implement all the steps if not needed

\* (could just leave to IdentityRanker if ranking is not needed).

\*

\* Theoretically, the actual implementation could override the above flow to add

\* more steps (e.g. add a transform step before ranking).

\* But it is recommended to add the additional steps into this base flow if the step proves

\* to have significant justification, or merge it into an existing step if it is a minor change.

\*

\* @tparam Target type of target request

\* @tparam Candidate type of candidate to return

\*/

trait RecommendationFlow[Target, Candidate <: UniversalNoun[Long]]

extends BaseRecommendationFlow[Target, Candidate]

with SideEffectsUtil[Target, Candidate] {

/\*\*

\* optionally update or enrich the request before executing the flows

\*/

protected def updateTarget(target: Target): Stitch[Target] = Stitch.value(target)

/\*\*

\* check if the target is eligible for the flow

\*/

protected def targetEligibility: Predicate[Target]

/\*\*

\* define the candidate sources that should be used for the given target

\*/

protected def candidateSources(target: Target): Seq[CandidateSource[Target, Candidate]]

/\*\*

\* filter invalid candidates before the ranking phase.

\*/

protected def preRankerCandidateFilter: Predicate[(Target, Candidate)]

/\*\*

\* rank the candidates

\*/

protected def selectRanker(target: Target): Ranker[Target, Candidate]

/\*\*

\* transform the candidates after ranking (e.g. dedupping, grouping and etc)

\*/

protected def postRankerTransform: Transform[Target, Candidate]

/\*\*

\* filter invalid candidates before returning the results.

\*

\* Some heavy filters e.g. SGS filter could be applied in this step

\*/

protected def validateCandidates: Predicate[(Target, Candidate)]

/\*\*

\* transform the candidates into results and return

\*/

protected def transformResults: Transform[Target, Candidate]

/\*\*

\* configuration for recommendation results

\*/

protected def resultsConfig(target: Target): RecommendationResultsConfig

/\*\*

\* track the quality factor the recommendation pipeline

\*/

protected def qualityFactorObserver: Option[QualityFactorObserver] = None

def statsReceiver: StatsReceiver

/\*\*

\* high level monitoring for the whole flow

\* (make sure to add monitoring for each individual component by yourself)

\*

\* additional candidates: count, stats, non\_empty\_count

\* target eligibility: latency, success, failures, request, count, valid\_count, invalid\_count, invalid\_reasons

\* candidate generation: latency, success, failures, request, count, non\_empty\_count, results\_stat

\* pre ranker filter: latency, success, failures, request, count, non\_empty\_count, results\_stat

\* ranker: latency, success, failures, request, count, non\_empty\_count, results\_stat

\* post ranker: latency, success, failures, request, count, non\_empty\_count, results\_stat

\* filter and take: latency, success, failures, request, count, non\_empty\_count, results\_stat, batch count

\* transform results: latency, success, failures, request, count, non\_empty\_count, results\_stat

\*/

import RecommendationFlow.\_

lazy val additionalCandidatesStats = statsReceiver.scope(AdditionalCandidatesStats)

lazy val targetEligibilityStats = statsReceiver.scope(TargetEligibilityStats)

lazy val candidateGenerationStats = statsReceiver.scope(CandidateGenerationStats)

lazy val preRankerFilterStats = statsReceiver.scope(PreRankerFilterStats)

lazy val rankerStats = statsReceiver.scope(RankerStats)

lazy val postRankerTransformStats = statsReceiver.scope(PostRankerTransformStats)

lazy val filterAndTakeStats = statsReceiver.scope(FilterAndTakeStats)

lazy val transformResultsStats = statsReceiver.scope(TransformResultsStats)

lazy val overallStats = statsReceiver.scope(OverallStats)

import StatsUtil.\_

override def process(

pipelineRequest: Target

): Stitch[RecommendationPipelineResult[Candidate, Seq[Candidate]]] = {

observeStitchQualityFactor(

profileStitchSeqResults(

updateTarget(pipelineRequest).flatMap { target =>

profilePredicateResult(targetEligibility(target), targetEligibilityStats).flatMap {

case PredicateResult.Valid => processValidTarget(target, Seq.empty)

case PredicateResult.Invalid(\_) => Stitch.Nil

}

},

overallStats

).map { candidates =>

RecommendationPipelineResult.empty.withResult(candidates)

},

qualityFactorObserver,

overallStats

)

}

protected def processValidTarget(

target: Target,

additionalCandidates: Seq[Candidate]

): Stitch[Seq[Candidate]] = {

/\*\*

\* A basic recommendation flow looks like this:

\*

\* 1. fetch candidates from candidate sources

\* 2. blend candidates with existing candidates

\* 3. filter the candidates (light filters) before ranking

\* 4. ranking

\* 5. filter and truncate the candidates using postRankerCandidateFilter

\* 6. transform the candidates based on product requirement

\*/

val candidateSourcesToFetch = candidateSources(target)

for {

candidates <- profileStitchSeqResults(

Stitch.traverse(candidateSourcesToFetch)(\_(target)).map(\_.flatten),

candidateGenerationStats

)

mergedCandidates =

profileSeqResults(additionalCandidates, additionalCandidatesStats) ++

candidates

filteredCandidates <- profileStitchSeqResults(

Predicate.filter(target, mergedCandidates, preRankerCandidateFilter),

preRankerFilterStats

)

rankedCandidates <- profileStitchSeqResults(

selectRanker(target).rank(target, filteredCandidates),

rankerStats

)

transformed <- profileStitchSeqResults(

postRankerTransform.transform(target, rankedCandidates),

postRankerTransformStats

)

truncated <- profileStitchSeqResults(

take(target, transformed, resultsConfig(target)),

filterAndTakeStats

)

results <- profileStitchSeqResults(

transformResults.transform(target, truncated),

transformResultsStats

)

\_ <- applySideEffects(

target,

candidateSourcesToFetch,

candidates,

mergedCandidates,

filteredCandidates,

rankedCandidates,

transformed,

truncated,

results)

} yield results

}

private[this] def take(

target: Target,

candidates: Seq[Candidate],

config: RecommendationResultsConfig

): Stitch[Seq[Candidate]] = {

Predicate

.batchFilterTake(

candidates.map(c => (target, c)),

validateCandidates,

config.batchForCandidatesCheck,

config.desiredCandidateCount,

statsReceiver

).map(\_.map(\_.\_2))

}

}

object RecommendationFlow {

val AdditionalCandidatesStats = "additional\_candidates"

val TargetEligibilityStats = "target\_eligibility"

val CandidateGenerationStats = "candidate\_generation"

val PreRankerFilterStats = "pre\_ranker\_filter"

val RankerStats = "ranker"

val PostRankerTransformStats = "post\_ranker\_transform"

val FilterAndTakeStats = "filter\_and\_take"

val TransformResultsStats = "transform\_results"

val OverallStats = "overall"

}