package com.twitter.product\_mixer.component\_library.selector

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

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

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

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

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

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

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

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

import scala.collection.mutable

object InsertAppendWeaveResults {

def apply[Query <: PipelineQuery, Bucket](

candidatePipelines: Set[CandidatePipelineIdentifier],

bucketer: Bucketer[Bucket],

): InsertAppendWeaveResults[Query, Bucket] =

new InsertAppendWeaveResults(SpecificPipelines(candidatePipelines), bucketer)

def apply[Query <: PipelineQuery, Bucket](

candidatePipeline: CandidatePipelineIdentifier,

bucketer: Bucketer[Bucket],

): InsertAppendWeaveResults[Query, Bucket] =

new InsertAppendWeaveResults(SpecificPipeline(candidatePipeline), bucketer)

}

/\*\*

\* Select candidates weave them together according to their [[Bucket]].

\*

\* Candidates are grouped according to [[Bucket]] and one candidate is added from each group until

\* no candidates belonging to any group are left.

\*

\* Functionally similar to [[InsertAppendPatternResults]]. [[InsertAppendPatternResults]] is useful

\* if you have more complex ordering requirements but it requires you to know all the buckets in

\* advance.

\*

\* @note The order in which candidates are weaved together depends on the order in which the buckets

\* were first seen on candidates.

\*

\* @example If the candidates are Seq(Tweet(10), Tweet(8), Tweet(3), Tweet(13)) and they are bucketed

\* using an IsEven bucketing function, then the resulting buckets would be:

\*

\* - Seq(Tweet(10), Tweet(8))

\* - Seq(Tweet(3), Tweet(13))

\*

\* The selector would then loop through these buckets and produce:

\*

\* - Tweet(10)

\* - Tweet(3)

\* - Tweet(8)

\* - Tweet(13)

\*

\* Note that first bucket encountered was the 'even' bucket so weaving proceeds first with

\* the even bucket then the odd bucket. Tweet(3) had been first then the opposite would be

\* true.

\*/

case class InsertAppendWeaveResults[-Query <: PipelineQuery, Bucket](

override val pipelineScope: CandidateScope,

bucketer: Bucketer[Bucket])

extends Selector[Query] {

override def apply(

query: Query,

remainingCandidates: Seq[CandidateWithDetails],

result: Seq[CandidateWithDetails]

): SelectorResult = {

val (bucketableCandidates, otherCandidates) =

remainingCandidates.partition(pipelineScope.contains)

val groupedCandidates = groupByBucket(bucketableCandidates)

val candidateBucketQueues: mutable.Queue[mutable.Queue[CandidateWithDetails]] =

mutable.Queue() ++= groupedCandidates

val newResult = mutable.ArrayBuffer[CandidateWithDetails]()

// Take the next group of candidates from the queue and attempt to add the first candidate from

// that group into the result. The loop will terminate when every queue is empty.

while (candidateBucketQueues.nonEmpty) {

val nextCandidateQueue = candidateBucketQueues.dequeue()

if (nextCandidateQueue.nonEmpty) {

newResult += nextCandidateQueue.dequeue()

// Re-queue this bucket of candidates if it's still non-empty

if (nextCandidateQueue.nonEmpty) {

candidateBucketQueues.enqueue(nextCandidateQueue)

}

}

}

SelectorResult(remainingCandidates = otherCandidates, result = result ++ newResult)

}

/\*\*

\* Similar to `groupBy` but respect the order in which individual bucket values are first seen.

\* This is useful when the candidates have already been sorted prior to the selector running.

\*/

private def groupByBucket(

candidates: Seq[CandidateWithDetails]

): mutable.ArrayBuffer[mutable.Queue[CandidateWithDetails]] = {

val bucketToCandidateGroupIndex = mutable.Map.empty[Bucket, Int]

val candidateGroups = mutable.ArrayBuffer[mutable.Queue[CandidateWithDetails]]()

candidates.foreach { candidate =>

val bucket = bucketer(candidate)

// Index points to the specific sub-group in candidateGroups where we want to insert the next

// candidate. If a bucket has already been seen then this value is known, otherwise we need

// to add a new entry for it.

if (!bucketToCandidateGroupIndex.contains(bucket)) {

candidateGroups.append(mutable.Queue())

bucketToCandidateGroupIndex.put(bucket, candidateGroups.length - 1)

}

candidateGroups(bucketToCandidateGroupIndex(bucket)).enqueue(candidate)

}

candidateGroups

}

}