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.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

/\*\*

\* Select candidates and add them according to the `pattern`.

\* The pattern is repeated until all candidates contained in the pattern are added to the `result`.

\* If the candidates for a specific [[Bucket]] in the pattern are exhausted, that [[Bucket]] will be

\* skipped on subsequent iterations.

\* If a candidate has a [[Bucket]] that isn't in the pattern it is added to the end of the `result`.

\* The end result is all candidates from all [[candidatePipelines]]s provided will end up in the result.

\*

\* @example If there are no more candidates from a given `CandidatePipeline` then it is skipped, so

\* with the pattern `Seq(A, A, B, C)`, if there are no more candidates from `B` then it is

\* effectively the same as `Seq(A, A, C)`. The `result` will contain all candidates from all

\* `CandidatePipeline`s who's `Bucket` is in the `pattern`.

\*

\* @example If the pattern is `Seq(A, A, B, C)` and the remaining candidates

\* from the provided `candidatePipelines` are:

\* - 5 `A`s

\* - 2 `B`s

\* - 1 `C`

\* - 1 `D`s

\*

\* then the resulting output for each iteration over the pattern is

\* - `Seq(A, A, B, C)`

\* - `Seq(A, A, B)` since there's no more `C`s

\* - `Seq(A)` since there are no more `B`s or `C`s

\* - `Seq(D)` since it wasn't in the pattern but is from one of the provided

\* `candidatePipelines`, it's appended at the end

\*

\* so the `result` that's returned would be `Seq(A, A, B, C, A, A, B, A, D)`

\*/

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

candidatePipelines: Set[CandidatePipelineIdentifier],

bucketer: Bucketer[Bucket],

pattern: Seq[Bucket])

extends Selector[Query] {

require(pattern.nonEmpty, "`pattern` must be non-empty")

override val pipelineScope: CandidateScope = SpecificPipelines(candidatePipelines)

private sealed trait PatternResult

private case object NotASelectedCandidatePipeline extends PatternResult

private case object NotABucketInThePattern extends PatternResult

private case class Bucketed(bucket: Bucket) extends PatternResult

private val allBucketsInPattern = pattern.toSet

override def apply(

query: Query,

remainingCandidates: Seq[CandidateWithDetails],

result: Seq[CandidateWithDetails]

): SelectorResult = {

val groupedCandidates: Map[PatternResult, Seq[CandidateWithDetails]] =

remainingCandidates.groupBy { candidateWithDetails =>

if (pipelineScope.contains(candidateWithDetails)) {

// if a candidate's Bucket doesnt appear in the pattern it's backfilled at the end

val bucket = bucketer(candidateWithDetails)

if (allBucketsInPattern.contains(bucket)) {

Bucketed(bucket)

} else {

NotABucketInThePattern

}

} else {

NotASelectedCandidatePipeline

}

}

val otherCandidates =

groupedCandidates.getOrElse(NotASelectedCandidatePipeline, Seq.empty)

val notABucketInThePattern =

groupedCandidates.getOrElse(NotABucketInThePattern, Seq.empty)

// mutable so we can remove finished iterators to optimize when looping for large patterns

val groupedBucketsIterators = mutable.HashMap(groupedCandidates.collect {

case (Bucketed(bucket), candidatesWithDetails) => (bucket, candidatesWithDetails.iterator)

}.toSeq: \_\*)

val patternIterator = Iterator.continually(pattern).flatten

val newResult = new mutable.ArrayBuffer[CandidateWithDetails]()

while (groupedBucketsIterators.nonEmpty) {

val bucket = patternIterator.next()

groupedBucketsIterators.get(bucket) match {

case Some(iterator) if iterator.nonEmpty => newResult += iterator.next()

case Some(\_) => groupedBucketsIterators.remove(bucket)

case None =>

}

}

SelectorResult(

remainingCandidates = otherCandidates,

result = result ++ newResult ++ notABucketInThePattern)

}

}