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

import com.twitter.product\_mixer.component\_library.model.candidate.CursorCandidate

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

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

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

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

import scala.collection.mutable

private[selector] object DropSelector {

/\*\*

\* Identify and merge duplicates using the supplied key extraction and merger functions. By default

\* this will keep only the first instance of a candidate in the `candidate` as determined by comparing

\* the contained candidate ID and class type. Subsequent matching instances will be dropped. For

\* more details, see DropSelector#dropDuplicates.

\*

\* @note [[com.twitter.product\_mixer.component\_library.model.candidate.CursorCandidate]] are ignored.

\* @note [[com.twitter.product\_mixer.core.model.common.presentation.ModuleCandidateWithDetails]] are ignored.

\*

\* @param candidates which may have elements to drop

\* @param duplicationKey how to generate a key for a candidate for identifying duplicates

\* @param mergeStrategy how to merge two candidates with the same key (by default pick the first one)

\*/

def dropDuplicates[Candidate <: CandidateWithDetails, Key](

pipelineScope: CandidateScope,

candidates: Seq[Candidate],

duplicationKey: DeduplicationKey[Key],

mergeStrategy: CandidateMergeStrategy

): Seq[Candidate] = {

val seenCandidatePositions = mutable.HashMap[Key, Int]()

// We assume that, most of the time, most candidates aren't duplicates so the result Seq will be

// approximately the size of the candidates Seq.

val deduplicatedCandidates = new mutable.ArrayBuffer[Candidate](candidates.length)

for (candidate <- candidates) {

candidate match {

// candidate is from one of the Pipelines the selector applies to and is not a CursorCandidate

case item: ItemCandidateWithDetails

if pipelineScope.contains(item) &&

!item.candidate.isInstanceOf[CursorCandidate] =>

val key = duplicationKey(item)

// Perform a merge if the candidate has been seen already

if (seenCandidatePositions.contains(key)) {

val candidateIndex = seenCandidatePositions(key)

// Safe because only ItemCandidateWithDetails are added to seenCandidatePositions so

// seenCandidatePositions(key) \*must\* point to an ItemCandidateWithDetails

val originalCandidate =

deduplicatedCandidates(candidateIndex).asInstanceOf[ItemCandidateWithDetails]

deduplicatedCandidates.update(

candidateIndex,

mergeStrategy(originalCandidate, item).asInstanceOf[Candidate])

} else {

// Otherwise add a new entry to the list of kept candidates and update our map to track

// the new index

deduplicatedCandidates.append(item.asInstanceOf[Candidate])

seenCandidatePositions.update(key, deduplicatedCandidates.length - 1)

}

case item => deduplicatedCandidates.append(item)

}

}

deduplicatedCandidates

}

/\*\*

\* Takes `candidates` from all [[CandidateWithDetails.source]]s but only `candidates` in the provided

\* `pipelineScope` are counted towards the `max` non-cursor candidates are included.

\*

\* @param max the maximum number of non-cursor candidates from the provided `pipelineScope` to return

\* @param candidates a sequence of candidates which may have elements dropped

\* @param pipelineScope the scope of which `candidates` should count towards the `max`

\*/

def takeUntil[Candidate <: CandidateWithDetails](

max: Int,

candidates: Seq[Candidate],

pipelineScope: CandidateScope = AllPipelines

): Seq[Candidate] = {

val resultsBuilder = Seq.newBuilder[Candidate]

resultsBuilder.sizeHint(candidates)

candidates.foldLeft(0) {

case (

count,

candidate @ ItemCandidateWithDetails(\_: CursorCandidate, \_, \_)

) =>

// keep cursors, not included in the `count`

resultsBuilder += candidate.asInstanceOf[Candidate]

count

case (count, candidate) if !pipelineScope.contains(candidate) =>

// keep candidates that don't match the provided `pipelineScope`, not included in the `count`

resultsBuilder += candidate

count

case (count, candidate) if count < max =>

// keep candidates if theres space and increment the `count`

resultsBuilder += candidate

count + 1

case (dropCurrentCandidate, \_) =>

// drop non-cursor candidate because theres no space left

dropCurrentCandidate

}

resultsBuilder.result()

}

}