package com.twitter.product\_mixer.core.service.selector\_executor

import com.twitter.finagle.stats.StatsReceiver

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.SelectorIdentifier

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

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

import com.twitter.product\_mixer.core.pipeline.pipeline\_failure.IllegalStateFailure

import com.twitter.product\_mixer.core.pipeline.pipeline\_failure.PipelineFailure

import com.twitter.product\_mixer.core.service.Executor

import com.twitter.stitch.Arrow

import javax.inject.Inject

import javax.inject.Singleton

/\*\*

\* Applies a `Seq[Selector]` in sequential order.

\* Returns the results, and also a detailed list each selector's results (for debugging / understandability).

\*/

@Singleton

class SelectorExecutor @Inject() (override val statsReceiver: StatsReceiver) extends Executor {

def arrow[Query <: PipelineQuery](

selectors: Seq[Selector[Query]],

context: Executor.Context

): Arrow[SelectorExecutor.Inputs[Query], SelectorExecutorResult] = {

if (selectors.isEmpty) {

throw PipelineFailure(

IllegalStateFailure,

"Must provide a non-empty Seq of Selectors. Check the config indicated by the componentStack and ensure that a non-empty Selector Seq is provided.",

componentStack = Some(context.componentStack)

)

}

val selectorArrows =

selectors.zipWithIndex.foldLeft(Arrow.identity[(Query, IndexedSeq[SelectorResult])]) {

case (previousSelectorArrows, (selector, index)) =>

val selectorResult = getIndividualSelectorIsoArrow(selector, index, context)

previousSelectorArrows.andThen(selectorResult)

}

Arrow

.zipWithArg(

Arrow

.map[SelectorExecutor.Inputs[Query], (Query, IndexedSeq[SelectorResult])] {

case SelectorExecutor.Inputs(query, candidates) =>

(query, IndexedSeq(SelectorResult(candidates, Seq.empty)))

}.andThen(selectorArrows)).map {

case (inputs, (\_, selectorResults)) =>

// the last results, safe because it's always non-empty since it starts with 1 element in it

val SelectorResult(remainingCandidates, result) = selectorResults.last

val resultsAndRemainingCandidates =

(result.iterator ++ remainingCandidates.iterator).toSet

// the droppedCandidates are all the candidates which are in neither the result or remainingCandidates

val droppedCandidates = inputs.candidatesWithDetails.iterator

.filterNot(resultsAndRemainingCandidates.contains)

.toIndexedSeq

SelectorExecutorResult(

selectedCandidates = result,

remainingCandidates = remainingCandidates,

droppedCandidates = droppedCandidates,

individualSelectorResults =

selectorResults.tail // `.tail` to remove the initial state we had

)

}

}

private def getIndividualSelectorIsoArrow[Query <: PipelineQuery](

selector: Selector[Query],

index: Int,

context: Executor.Context

): Arrow.Iso[(Query, IndexedSeq[SelectorResult])] = {

val identifier = SelectorIdentifier(selector.getClass.getSimpleName, index)

val arrow = Arrow

.identity[(Query, IndexedSeq[SelectorResult])]

.map {

case (query, previousResults) =>

// last is safe here because we pass in a non-empty IndexedSeq

val previousResult = previousResults.last

val currentResult = selector.apply(

query,

previousResult.remainingCandidates,

previousResult.result

)

(query, previousResults :+ currentResult)

}

wrapComponentsWithTracingOnly(context, identifier)(

wrapWithErrorHandling(context, identifier)(

arrow

)

)

}

}

object SelectorExecutor {

case class Inputs[Query <: PipelineQuery](

query: Query,

candidatesWithDetails: Seq[CandidateWithDetails])

}