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

import com.twitter.goldfinch.api.AdsInjectionSurfaceAreas.SurfaceAreaName

import com.twitter.goldfinch.api.AdsInjectorAdditionalRequestParams

import com.twitter.goldfinch.api.AdsInjectorOutput

import com.twitter.goldfinch.api.{AdsInjector => GoldfinchAdsInjector}

import com.twitter.product\_mixer.component\_library.model.query.ads.\_

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

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

import CandidateScope.PartitionedCandidates

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

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

/\*\*

\* Injects the sequence of AdCandidates in the `result` in the

\* sequence of the Other Candidates(which are not ads).

\*

\* Every SurfaceArea or DisplayLocation runs their own desired set of adjusters(set in pipeline)

\* to inject ads and reposition the ads in the sequence of other candidates of `result` :

\* which are fetched by AdsInjectionSurfaceAreaAdjustersMap

\* Note: The original sequence of non\_promoted entries(non-ads) is retained and the ads

\* are inserted in the sequence using `goldfinch` library based on the 'insertion-position'

\* hydrated in AdsCandidate by Adserver/Admixer.

\*

\* \*\*\*\*\* Goldfinch recommends to run this selector as close to the marshalling of candidates to have

\* more realistic view of served-timeline in Goldfinch-BQ-Logs and avoid any further updates on the

\* timeline(sequence of entries) created. \*\*\*\*

\*

\* Any surface area like `search\_tweets(surface\_area)` can call

\* InsertAdResults(surfaceArea = "TweetSearch", candidatePipeline = adsCandidatePipeline.identifier,

\* ProductMixerAdsInjector = productMixerAdsInjector)

\* where the pipeline config can call

\* productMixerAdsInjector.forSurfaceArea("TweetSearch") to get AdsInjector Object

\*

\* @example

\* `Seq(source1NonAd\_Id1, source1NonAd\_Id2, source2NonAd\_Id1, source2NonAd\_Id2,source1NonAd\_Id3, source3NonAd\_Id3,source3Ad\_Id1\_InsertionPos1, source3Ad\_Id2\_InsertionPos4)`

\* then the output result can be

\* `Seq(source1NonAd\_Id1, source3Ad\_Id1\_InsertionPos1, source1NonAd\_Id2, source2NonAd\_Id1, source3Ad\_Id2\_InsertionPos4,source2NonAd\_Id2, source1NonAd\_Id3, source3NonAd\_Id3)`

\* depending on the insertion position of Ads and other adjusters shifting the ads

\*/

case class InsertAdResults(

surfaceAreaName: SurfaceAreaName,

adsInjector: GoldfinchAdsInjector[

PipelineQuery with AdsQuery,

CandidateWithDetails,

CandidateWithDetails

],

adsCandidatePipeline: CandidatePipelineIdentifier)

extends Selector[PipelineQuery with AdsQuery] {

override val pipelineScope: CandidateScope = SpecificPipeline(adsCandidatePipeline)

override def apply(

query: PipelineQuery with AdsQuery,

remainingCandidates: Seq[CandidateWithDetails],

result: Seq[CandidateWithDetails]

): SelectorResult = {

// Read into ads and non-ads candidates.

val PartitionedCandidates(adCandidates, otherRemainingCandidates) =

pipelineScope.partition(remainingCandidates)

// Create this param from Query/AdsCandidate based on surface\_area, if required.

val adsInjectorAdditionalRequestParams =

AdsInjectorAdditionalRequestParams(budgetAwareExperimentId = None)

val adsInjectorOutput: AdsInjectorOutput[CandidateWithDetails, CandidateWithDetails] =

adsInjector.applyForAllEntries(

query = query,

nonPromotedEntries = result,

promotedEntries = adCandidates,

adsInjectorAdditionalRequestParams = adsInjectorAdditionalRequestParams)

val updatedRemainingCandidates = otherRemainingCandidates ++

GoldfinchResults(adsInjectorOutput.unusedEntries).adapt

val mergedResults = GoldfinchResults(adsInjectorOutput.mergedEntries).adapt

SelectorResult(remainingCandidates = updatedRemainingCandidates, result = mergedResults)

}

/\*\*

\* Goldfinch separates NonPromotedEntryType and PromotedEntryType models, while in ProMix both

\* non-promoted and promoted entries are CandidateWithDetails. As such, we need to flatten the

\* result back into a single Seq of CandidateWithDetails. See [[AdsInjectorOutput]]

\*/

case class GoldfinchResults(results: Seq[Either[CandidateWithDetails, CandidateWithDetails]]) {

def adapt: Seq[CandidateWithDetails] = {

results.collect {

case Right(value) => value

case Left(value) => value

}

}

}

}