package com.twitter.follow\_recommendations.common.transforms.tracking\_token

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.follow\_recommendations.common.base.Transform

import com.twitter.follow\_recommendations.common.models.CandidateUser

import com.twitter.follow\_recommendations.common.models.HasDisplayLocation

import com.twitter.follow\_recommendations.common.models.Session

import com.twitter.follow\_recommendations.common.models.TrackingToken

import com.twitter.hermit.constants.AlgorithmFeedbackTokens.AlgorithmToFeedbackTokenMap

import com.twitter.hermit.model.Algorithm

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

import com.twitter.product\_mixer.core.model.marshalling.request.HasClientContext

import com.twitter.stitch.Stitch

import com.twitter.util.logging.Logging

import javax.inject.Inject

import javax.inject.Singleton

/\*\*

\* This transform adds the tracking token for all candidates

\* Since this happens in the same request, we use the same trace-id for all candidates

\* There are no RPC calls in this transform so it's safe to chain it with `andThen` at the end of

\* all other product-specific transforms

\*/

@Singleton

class TrackingTokenTransform @Inject() (baseStatsReceiver: StatsReceiver)

extends Transform[HasDisplayLocation with HasClientContext, CandidateUser]

with Logging {

def profileResults(

target: HasDisplayLocation with HasClientContext,

candidates: Seq[CandidateUser]

) = {

// Metrics to track # results per candidate source

val stats = baseStatsReceiver.scope(target.displayLocation.toString + "/final\_results")

stats.stat("total").add(candidates.size)

stats.counter(target.displayLocation.toString).incr()

val flattenedCandidates: Seq[(CandidateSourceIdentifier, CandidateUser)] = for {

candidate <- candidates

identifier <- candidate.getPrimaryCandidateSource

} yield (identifier, candidate)

val candidatesGroupedBySource: Map[CandidateSourceIdentifier, Seq[CandidateUser]] =

flattenedCandidates.groupBy(\_.\_1).mapValues(\_.map(\_.\_2))

candidatesGroupedBySource map {

case (source, candidates) => stats.stat(source.name).add(candidates.size)

}

}

override def transform(

target: HasDisplayLocation with HasClientContext,

candidates: Seq[CandidateUser]

): Stitch[Seq[CandidateUser]] = {

profileResults(target, candidates)

Stitch.value(

target.getOptionalUserId

.map { \_ =>

candidates.map {

candidate =>

val token = Some(TrackingToken(

sessionId = Session.getSessionId,

displayLocation = Some(target.displayLocation),

controllerData = None,

algorithmId = candidate.userCandidateSourceDetails.flatMap(\_.primaryCandidateSource

.flatMap { identifier =>

Algorithm.withNameOpt(identifier.name).flatMap(AlgorithmToFeedbackTokenMap.get)

})

))

candidate.copy(trackingToken = token)

}

}.getOrElse(candidates))

}

}