package com.twitter.follow\_recommendations.common.models

import com.twitter.follow\_recommendations.logging.{thriftscala => offline}

import com.twitter.follow\_recommendations.{thriftscala => t}

import com.twitter.hermit.constants.AlgorithmFeedbackTokens.\_

import com.twitter.hermit.ml.models.Feature

import com.twitter.hermit.model.Algorithm

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

/\*\*

\* primaryCandidateSource param is showing the candidate source that responsible for generating this

\* candidate, as the candidate might have gone through multiple candidate sources to get generated

\* (for example if it has generated by a composite source). WeightedCandidateSourceRanker uses this

\* field to do the sampling over candidate sources. All the sources used for generating this

\* candidate (including the primary source) and their corresponding score exist in the

\* candidateSourceScores field.

\*/

case class UserCandidateSourceDetails(

primaryCandidateSource: Option[CandidateSourceIdentifier],

candidateSourceScores: Map[CandidateSourceIdentifier, Option[Double]] = Map.empty,

candidateSourceRanks: Map[CandidateSourceIdentifier, Int] = Map.empty,

addressBookMetadata: Option[AddressBookMetadata] = None,

candidateSourceFeatures: Map[CandidateSourceIdentifier, Seq[Feature]] = Map.empty,

) {

def toThrift: t.CandidateSourceDetails = {

t.CandidateSourceDetails(

candidateSourceScores = Some(candidateSourceScores.map {

case (identifier, score) =>

(identifier.name, score.getOrElse(0.0d))

}),

primarySource = for {

identifier <- primaryCandidateSource

algo <- Algorithm.withNameOpt(identifier.name)

feedbackToken <- AlgorithmToFeedbackTokenMap.get(algo)

} yield feedbackToken

)

}

def toOfflineThrift: offline.CandidateSourceDetails = {

offline.CandidateSourceDetails(

candidateSourceScores = Some(candidateSourceScores.map {

case (identifier, score) =>

(identifier.name, score.getOrElse(0.0d))

}),

primarySource = for {

identifier <- primaryCandidateSource

algo <- Algorithm.withNameOpt(identifier.name)

feedbackToken <- AlgorithmToFeedbackTokenMap.get(algo)

} yield feedbackToken

)

}

}

object UserCandidateSourceDetails {

val algorithmNameMap: Map[String, Algorithm.Value] = Algorithm.values.map {

algorithmValue: Algorithm.Value =>

(algorithmValue.toString, algorithmValue)

}.toMap

/\*\*

\* This method is used to parse the candidate source of the candidates, which is only passed from

\* the scoreUserCandidates endpoint. We create custom candidate source identifiers which

\* CandidateAlgorithmSource will read from to hydrate the algorithm id feature.

\* candidateSourceScores will not be populated from the endpoint, but we add the conversion for

\* completeness. Note that the conversion uses the raw string of the Algorithm rather than the

\* assigned strings that we give to our own candidate sources in the FRS.

\*/

def fromThrift(details: t.CandidateSourceDetails): UserCandidateSourceDetails = {

val primaryCandidateSource: Option[CandidateSourceIdentifier] = for {

primarySourceToken <- details.primarySource

algo <- TokenToAlgorithmMap.get(primarySourceToken)

} yield CandidateSourceIdentifier(algo.toString)

val candidateSourceScores = for {

scoreMap <- details.candidateSourceScores.toSeq

(name, score) <- scoreMap

algo <- algorithmNameMap.get(name)

} yield {

CandidateSourceIdentifier(algo.toString) -> Some(score)

}

val candidateSourceRanks = for {

rankMap <- details.candidateSourceRanks.toSeq

(name, rank) <- rankMap

algo <- algorithmNameMap.get(name)

} yield {

CandidateSourceIdentifier(algo.toString) -> rank

}

UserCandidateSourceDetails(

primaryCandidateSource = primaryCandidateSource,

candidateSourceScores = candidateSourceScores.toMap,

candidateSourceRanks = candidateSourceRanks.toMap,

addressBookMetadata = None,

candidateSourceFeatures = Map.empty

)

}

}