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.ml.api.thriftscala.{DataRecord => TDataRecord}

import com.twitter.ml.api.util.ScalaToJavaDataRecordConversions

import com.twitter.timelines.configapi.HasParams

import com.twitter.timelines.configapi.Params

import com.twitter.product\_mixer.core.model.common.UniversalNoun

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

trait FollowableEntity extends UniversalNoun[Long]

trait Recommendation

extends FollowableEntity

with HasReason

with HasAdMetadata

with HasTrackingToken {

val score: Option[Double]

def toThrift: t.Recommendation

def toOfflineThrift: offline.OfflineRecommendation

}

case class CandidateUser(

override val id: Long,

override val score: Option[Double] = None,

override val reason: Option[Reason] = None,

override val userCandidateSourceDetails: Option[UserCandidateSourceDetails] = None,

override val adMetadata: Option[AdMetadata] = None,

override val trackingToken: Option[TrackingToken] = None,

override val dataRecord: Option[RichDataRecord] = None,

override val scores: Option[Scores] = None,

override val infoPerRankingStage: Option[scala.collection.Map[String, RankingInfo]] = None,

override val params: Params = Params.Invalid,

override val engagements: Seq[EngagementType] = Nil,

override val recommendationFlowIdentifier: Option[String] = None)

extends Recommendation

with HasUserCandidateSourceDetails

with HasDataRecord

with HasScores

with HasParams

with HasEngagements

with HasRecommendationFlowIdentifier

with HasInfoPerRankingStage {

val rankerIdsStr: Option[Seq[String]] = {

val strs = scores.map(\_.scores.flatMap(\_.rankerId.map(\_.toString)))

if (strs.exists(\_.nonEmpty)) strs else None

}

val thriftDataRecord: Option[TDataRecord] = for {

richDataRecord <- dataRecord

dr <- richDataRecord.dataRecord

} yield {

ScalaToJavaDataRecordConversions.javaDataRecord2ScalaDataRecord(dr)

}

val toOfflineUserThrift: offline.OfflineUserRecommendation = {

val scoringDetails =

if (userCandidateSourceDetails.isEmpty && score.isEmpty && thriftDataRecord.isEmpty) {

None

} else {

Some(

offline.ScoringDetails(

candidateSourceDetails = userCandidateSourceDetails.map(\_.toOfflineThrift),

score = score,

dataRecord = thriftDataRecord,

rankerIds = rankerIdsStr,

infoPerRankingStage = infoPerRankingStage.map(\_.mapValues(\_.toOfflineThrift))

)

)

}

offline

.OfflineUserRecommendation(

id,

reason.map(\_.toOfflineThrift),

adMetadata.map(\_.adImpression),

trackingToken.map(\_.toOfflineThrift),

scoringDetails = scoringDetails

)

}

override val toOfflineThrift: offline.OfflineRecommendation =

offline.OfflineRecommendation.User(toOfflineUserThrift)

val toUserThrift: t.UserRecommendation = {

val scoringDetails =

if (userCandidateSourceDetails.isEmpty && score.isEmpty && thriftDataRecord.isEmpty && scores.isEmpty) {

None

} else {

Some(

t.ScoringDetails(

candidateSourceDetails = userCandidateSourceDetails.map(\_.toThrift),

score = score,

dataRecord = thriftDataRecord,

rankerIds = rankerIdsStr,

debugDataRecord = dataRecord.flatMap(\_.debugDataRecord),

infoPerRankingStage = infoPerRankingStage.map(\_.mapValues(\_.toThrift))

)

)

}

t.UserRecommendation(

userId = id,

reason = reason.map(\_.toThrift),

adImpression = adMetadata.map(\_.adImpression),

trackingInfo = trackingToken.map(TrackingToken.serialize),

scoringDetails = scoringDetails,

recommendationFlowIdentifier = recommendationFlowIdentifier

)

}

override val toThrift: t.Recommendation =

t.Recommendation.User(toUserThrift)

def setFollowProof(followProofOpt: Option[FollowProof]): CandidateUser = {

this.copy(

reason = reason

.map { reason =>

reason.copy(

accountProof = reason.accountProof

.map { accountProof =>

accountProof.copy(followProof = followProofOpt)

}.orElse(Some(AccountProof(followProof = followProofOpt)))

)

}.orElse(Some(Reason(Some(AccountProof(followProof = followProofOpt)))))

)

}

def addScore(score: Score): CandidateUser = {

val newScores = scores match {

case Some(existingScores) => existingScores.copy(scores = existingScores.scores :+ score)

case None => Scores(Seq(score))

}

this.copy(scores = Some(newScores))

}

}

object CandidateUser {

val DefaultCandidateScore = 1.0

// for converting candidate in ScoringUserRequest

def fromUserRecommendation(candidate: t.UserRecommendation): CandidateUser = {

// we only use the primary candidate source for now

val userCandidateSourceDetails = for {

scoringDetails <- candidate.scoringDetails

candidateSourceDetails <- scoringDetails.candidateSourceDetails

} yield UserCandidateSourceDetails(

primaryCandidateSource = candidateSourceDetails.primarySource

.flatMap(AlgorithmFeedbackTokens.TokenToAlgorithmMap.get).map { algo =>

CandidateSourceIdentifier(algo.toString)

},

candidateSourceScores = fromThriftScoreMap(candidateSourceDetails.candidateSourceScores),

candidateSourceRanks = fromThriftRankMap(candidateSourceDetails.candidateSourceRanks),

addressBookMetadata = None

)

CandidateUser(

id = candidate.userId,

score = candidate.scoringDetails.flatMap(\_.score),

reason = candidate.reason.map(Reason.fromThrift),

userCandidateSourceDetails = userCandidateSourceDetails,

trackingToken = candidate.trackingInfo.map(TrackingToken.deserialize),

recommendationFlowIdentifier = candidate.recommendationFlowIdentifier,

infoPerRankingStage = candidate.scoringDetails.flatMap(

\_.infoPerRankingStage.map(\_.mapValues(RankingInfo.fromThrift)))

)

}

def fromThriftScoreMap(

thriftMapOpt: Option[scala.collection.Map[String, Double]]

): Map[CandidateSourceIdentifier, Option[Double]] = {

(for {

thriftMap <- thriftMapOpt.toSeq

(algoName, score) <- thriftMap.toSeq

} yield {

CandidateSourceIdentifier(algoName) -> Some(score)

}).toMap

}

def fromThriftRankMap(

thriftMapOpt: Option[scala.collection.Map[String, Int]]

): Map[CandidateSourceIdentifier, Int] = {

(for {

thriftMap <- thriftMapOpt.toSeq

(algoName, rank) <- thriftMap.toSeq

} yield {

CandidateSourceIdentifier(algoName) -> rank

}).toMap

}

}