package com.twitter.frigate.pushservice.scriber

import com.twitter.bijection.Base64String

import com.twitter.bijection.Injection

import com.twitter.bijection.scrooge.BinaryScalaCodec

import com.twitter.core\_workflows.user\_model.thriftscala.{UserState => ThriftUserState}

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.tracing.Trace

import com.twitter.frigate.common.base.CandidateDetails

import com.twitter.frigate.common.base.CandidateResult

import com.twitter.frigate.common.base.Invalid

import com.twitter.frigate.common.base.OK

import com.twitter.frigate.common.base.Result

import com.twitter.frigate.common.rec\_types.RecTypes

import com.twitter.frigate.data\_pipeline.features\_common.PushQualityModelFeatureContext

import com.twitter.frigate.pushservice.model.PushTypes.PushCandidate

import com.twitter.frigate.pushservice.model.PushTypes.Target

import com.twitter.frigate.pushservice.params.PushFeatureSwitchParams

import com.twitter.frigate.pushservice.params.PushParams

import com.twitter.frigate.scribe.thriftscala.CandidateFilteredOutStep

import com.twitter.frigate.scribe.thriftscala.CandidateRequestInfo

import com.twitter.frigate.scribe.thriftscala.MrRequestScribe

import com.twitter.frigate.scribe.thriftscala.TargetUserInfo

import com.twitter.frigate.thriftscala.FrigateNotification

import com.twitter.frigate.thriftscala.TweetNotification

import com.twitter.frigate.thriftscala.{SocialContextAction => TSocialContextAction}

import com.twitter.logging.Logger

import com.twitter.ml.api.DataRecord

import com.twitter.ml.api.Feature

import com.twitter.ml.api.FeatureType

import com.twitter.ml.api.util.SRichDataRecord

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

import com.twitter.nrel.heavyranker.PushPredictionHelper

import com.twitter.util.Future

import com.twitter.util.Time

import java.util.UUID

import scala.collection.mutable

class MrRequestScribeHandler(mrRequestScriberNode: String, stats: StatsReceiver) {

private val mrRequestScribeLogger = Logger(mrRequestScriberNode)

private val mrRequestScribeTargetFilteringStats =

stats.counter("MrRequestScribeHandler\_target\_filtering")

private val mrRequestScribeCandidateFilteringStats =

stats.counter("MrRequestScribeHandler\_candidate\_filtering")

private val mrRequestScribeInvalidStats =

stats.counter("MrRequestScribeHandler\_invalid\_filtering")

private val mrRequestScribeUnsupportedFeatureTypeStats =

stats.counter("MrRequestScribeHandler\_unsupported\_feature\_type")

private val mrRequestScribeNotIncludedFeatureStats =

stats.counter("MrRequestScribeHandler\_not\_included\_features")

private final val MrRequestScribeInjection: Injection[MrRequestScribe, String] = BinaryScalaCodec(

MrRequestScribe

) andThen Injection.connect[Array[Byte], Base64String, String]

/\*\*

\*

\* @param target : Target user id

\* @param result : Result for target filtering

\*

\* @return

\*/

def scribeForTargetFiltering(target: Target, result: Result): Future[Option[MrRequestScribe]] = {

if (target.isLoggedOutUser || !enableTargetFilteringScribing(target)) {

Future.None

} else {

val predicate = result match {

case Invalid(reason) => reason

case \_ =>

mrRequestScribeInvalidStats.incr()

throw new IllegalStateException("Invalid reason for Target Filtering " + result)

}

buildScribeThrift(target, predicate, None).map { targetFilteredScribe =>

writeAtTargetFilteringStep(target, targetFilteredScribe)

Some(targetFilteredScribe)

}

}

}

/\*\*

\*

\* @param target : Target user id

\* @param hydratedCandidates : Candidates hydrated with details: impressionId, frigateNotification and source

\* @param preRankingFilteredCandidates : Candidates result filtered out at preRanking filtering step

\* @param rankedCandidates : Sorted candidates details ranked by ranking step

\* @param rerankedCandidates : Sorted candidates details ranked by reranking step

\* @param restrictFilteredCandidates : Candidates details filtered out at restrict step

\* @param allTakeCandidateResults : Candidates results at take step, include the candidates we take and the candidates filtered out at take step [with different result]

\*

\* @return

\*/

def scribeForCandidateFiltering(

target: Target,

hydratedCandidates: Seq[CandidateDetails[PushCandidate]],

preRankingFilteredCandidates: Seq[CandidateResult[PushCandidate, Result]],

rankedCandidates: Seq[CandidateDetails[PushCandidate]],

rerankedCandidates: Seq[CandidateDetails[PushCandidate]],

restrictFilteredCandidates: Seq[CandidateDetails[PushCandidate]],

allTakeCandidateResults: Seq[CandidateResult[PushCandidate, Result]]

): Future[Seq[MrRequestScribe]] = {

if (target.isLoggedOutUser || target.isEmailUser) {

Future.Nil

} else if (enableCandidateFilteringScribing(target)) {

val hydrateFeature =

target.params(PushFeatureSwitchParams.EnableMrRequestScribingWithFeatureHydrating) ||

target.scribeFeatureForRequestScribe

val candidateRequestInfoSeq = generateCandidatesScribeInfo(

hydratedCandidates,

preRankingFilteredCandidates,

rankedCandidates,

rerankedCandidates,

restrictFilteredCandidates,

allTakeCandidateResults,

isFeatureHydratingEnabled = hydrateFeature

)

val flattenStructure =

target.params(PushFeatureSwitchParams.EnableFlattenMrRequestScribing) || hydrateFeature

candidateRequestInfoSeq.flatMap { candidateRequestInfos =>

if (flattenStructure) {

Future.collect {

candidateRequestInfos.map { candidateRequestInfo =>

buildScribeThrift(target, None, Some(Seq(candidateRequestInfo)))

.map { mrRequestScribe =>

writeAtCandidateFilteringStep(target, mrRequestScribe)

mrRequestScribe

}

}

}

} else {

buildScribeThrift(target, None, Some(candidateRequestInfos))

.map { mrRequestScribe =>

writeAtCandidateFilteringStep(target, mrRequestScribe)

Seq(mrRequestScribe)

}

}

}

} else Future.Nil

}

private def buildScribeThrift(

target: Target,

targetFilteredOutPredicate: Option[String],

candidatesRequestInfo: Option[Seq[CandidateRequestInfo]]

): Future[MrRequestScribe] = {

Future

.join(

target.targetUserState,

generateTargetFeatureScribeInfo(target),

target.targetUser).map {

case (userStateOption, targetFeatureOption, gizmoduckUserOpt) =>

val userState = userStateOption.map(userState => ThriftUserState(userState.id))

val targetFeatures =

targetFeatureOption.map(ScalaToJavaDataRecordConversions.javaDataRecord2ScalaDataRecord)

val traceId = Trace.id.traceId.toLong

MrRequestScribe(

requestId = UUID.randomUUID.toString.replaceAll("-", ""),

scribedTimeMs = Time.now.inMilliseconds,

targetUserId = target.targetId,

targetUserInfo = Some(

TargetUserInfo(

userState,

features = targetFeatures,

userType = gizmoduckUserOpt.map(\_.userType))

),

targetFilteredOutPredicate = targetFilteredOutPredicate,

candidates = candidatesRequestInfo,

traceId = Some(traceId)

)

}

}

private def generateTargetFeatureScribeInfo(

target: Target

): Future[Option[DataRecord]] = {

val featureList =

target.params(PushFeatureSwitchParams.TargetLevelFeatureListForMrRequestScribing)

if (featureList.nonEmpty) {

PushPredictionHelper

.getDataRecordFromTargetFeatureMap(

target.targetId,

target.featureMap,

stats

).map { dataRecord =>

val richRecord =

new SRichDataRecord(dataRecord, PushQualityModelFeatureContext.featureContext)

val selectedRecord =

SRichDataRecord(new DataRecord(), PushQualityModelFeatureContext.featureContext)

featureList.map { featureName =>

val feature: Feature[\_] = {

try {

PushQualityModelFeatureContext.featureContext.getFeature(featureName)

} catch {

case \_: Exception =>

mrRequestScribeNotIncludedFeatureStats.incr()

throw new IllegalStateException(

"Scribing features not included in FeatureContext: " + featureName)

}

}

richRecord.getFeatureValueOpt(feature).foreach { featureVal =>

feature.getFeatureType() match {

case FeatureType.BINARY =>

selectedRecord.setFeatureValue(

feature.asInstanceOf[Feature[Boolean]],

featureVal.asInstanceOf[Boolean])

case FeatureType.CONTINUOUS =>

selectedRecord.setFeatureValue(

feature.asInstanceOf[Feature[Double]],

featureVal.asInstanceOf[Double])

case FeatureType.STRING =>

selectedRecord.setFeatureValue(

feature.asInstanceOf[Feature[String]],

featureVal.asInstanceOf[String])

case FeatureType.DISCRETE =>

selectedRecord.setFeatureValue(

feature.asInstanceOf[Feature[Long]],

featureVal.asInstanceOf[Long])

case \_ =>

mrRequestScribeUnsupportedFeatureTypeStats.incr()

}

}

}

Some(selectedRecord.getRecord)

}

} else Future.None

}

private def generateCandidatesScribeInfo(

hydratedCandidates: Seq[CandidateDetails[PushCandidate]],

preRankingFilteredCandidates: Seq[CandidateResult[PushCandidate, Result]],

rankedCandidates: Seq[CandidateDetails[PushCandidate]],

rerankedCandidates: Seq[CandidateDetails[PushCandidate]],

restrictFilteredCandidates: Seq[CandidateDetails[PushCandidate]],

allTakeCandidateResults: Seq[CandidateResult[PushCandidate, Result]],

isFeatureHydratingEnabled: Boolean

): Future[Seq[CandidateRequestInfo]] = {

val candidatesMap = new mutable.HashMap[String, CandidateRequestInfo]

hydratedCandidates.foreach { hydratedCandidate =>

val frgNotif = hydratedCandidate.candidate.frigateNotification

val simplifiedTweetNotificationOpt = frgNotif.tweetNotification.map { tweetNotification =>

TweetNotification(

tweetNotification.tweetId,

Seq.empty[TSocialContextAction],

tweetNotification.tweetAuthorId)

}

val simplifiedFrigateNotification = FrigateNotification(

frgNotif.commonRecommendationType,

frgNotif.notificationDisplayLocation,

tweetNotification = simplifiedTweetNotificationOpt

)

candidatesMap(hydratedCandidate.candidate.impressionId) = CandidateRequestInfo(

candidateId = "",

candidateSource = hydratedCandidate.source.substring(

0,

Math.min(6, hydratedCandidate.source.length)

),

frigateNotification = Some(simplifiedFrigateNotification),

modelScore = None,

rankPosition = None,

rerankPosition = None,

features = None,

isSent = Some(false)

)

}

preRankingFilteredCandidates.foreach { preRankingFilteredCandidateResult =>

candidatesMap(preRankingFilteredCandidateResult.candidate.impressionId) =

candidatesMap(preRankingFilteredCandidateResult.candidate.impressionId)

.copy(

candidateFilteredOutPredicate = preRankingFilteredCandidateResult.result match {

case Invalid(reason) => reason

case \_ => {

mrRequestScribeInvalidStats.incr()

throw new IllegalStateException(

"Invalid reason for Candidate Filtering " + preRankingFilteredCandidateResult.result)

}

},

candidateFilteredOutStep = Some(CandidateFilteredOutStep.PreRankFiltering)

)

}

for {

\_ <- Future.collectToTry {

rankedCandidates.zipWithIndex.map {

case (rankedCandidateDetail, index) =>

val modelScoresFut = {

val crt = rankedCandidateDetail.candidate.commonRecType

if (RecTypes.notEligibleForModelScoreTracking.contains(crt)) Future.None

else rankedCandidateDetail.candidate.modelScores.map(Some(\_))

}

modelScoresFut.map { modelScores =>

candidatesMap(rankedCandidateDetail.candidate.impressionId) =

candidatesMap(rankedCandidateDetail.candidate.impressionId).copy(

rankPosition = Some(index),

modelScore = modelScores

)

}

}

}

\_ = rerankedCandidates.zipWithIndex.foreach {

case (rerankedCandidateDetail, index) => {

candidatesMap(rerankedCandidateDetail.candidate.impressionId) =

candidatesMap(rerankedCandidateDetail.candidate.impressionId).copy(

rerankPosition = Some(index)

)

}

}

\_ <- Future.collectToTry {

rerankedCandidates.map { rerankedCandidateDetail =>

if (isFeatureHydratingEnabled) {

PushPredictionHelper

.getDataRecord(

rerankedCandidateDetail.candidate.target.targetHydrationContext,

rerankedCandidateDetail.candidate.target.featureMap,

rerankedCandidateDetail.candidate.candidateHydrationContext,

rerankedCandidateDetail.candidate.candidateFeatureMap(),

stats

).map { features =>

candidatesMap(rerankedCandidateDetail.candidate.impressionId) =

candidatesMap(rerankedCandidateDetail.candidate.impressionId).copy(

features = Some(

ScalaToJavaDataRecordConversions.javaDataRecord2ScalaDataRecord(features))

)

}

} else Future.Unit

}

}

\_ = restrictFilteredCandidates.foreach { restrictFilteredCandidateDetatil =>

candidatesMap(restrictFilteredCandidateDetatil.candidate.impressionId) =

candidatesMap(restrictFilteredCandidateDetatil.candidate.impressionId)

.copy(candidateFilteredOutStep = Some(CandidateFilteredOutStep.Restrict))

}

\_ = allTakeCandidateResults.foreach { allTakeCandidateResult =>

allTakeCandidateResult.result match {

case OK =>

candidatesMap(allTakeCandidateResult.candidate.impressionId) =

candidatesMap(allTakeCandidateResult.candidate.impressionId).copy(isSent = Some(true))

case Invalid(reason) =>

candidatesMap(allTakeCandidateResult.candidate.impressionId) =

candidatesMap(allTakeCandidateResult.candidate.impressionId).copy(

candidateFilteredOutPredicate = reason,

candidateFilteredOutStep = Some(CandidateFilteredOutStep.PostRankFiltering))

case \_ =>

mrRequestScribeInvalidStats.incr()

throw new IllegalStateException(

"Invalid reason for Candidate Filtering " + allTakeCandidateResult.result)

}

}

} yield candidatesMap.values.toSeq

}

private def enableTargetFilteringScribing(target: Target): Boolean = {

target.params(PushParams.EnableMrRequestScribing) && target.params(

PushFeatureSwitchParams.EnableMrRequestScribingForTargetFiltering)

}

private def enableCandidateFilteringScribing(target: Target): Boolean = {

target.params(PushParams.EnableMrRequestScribing) && target.params(

PushFeatureSwitchParams.EnableMrRequestScribingForCandidateFiltering)

}

private def writeAtTargetFilteringStep(target: Target, mrRequestScribe: MrRequestScribe) = {

logToScribe(mrRequestScribe)

mrRequestScribeTargetFilteringStats.incr()

}

private def writeAtCandidateFilteringStep(target: Target, mrRequestScribe: MrRequestScribe) = {

logToScribe(mrRequestScribe)

mrRequestScribeCandidateFilteringStats.incr()

}

private def logToScribe(mrRequestScribe: MrRequestScribe): Unit = {

val logEntry: String = MrRequestScribeInjection(mrRequestScribe)

mrRequestScribeLogger.info(logEntry)

}

}