package com.twitter.cr\_mixer.controller

import com.twitter.core\_workflows.user\_model.thriftscala.UserState

import com.twitter.cr\_mixer.candidate\_generation.AdsCandidateGenerator

import com.twitter.cr\_mixer.candidate\_generation.CrCandidateGenerator

import com.twitter.cr\_mixer.candidate\_generation.FrsTweetCandidateGenerator

import com.twitter.cr\_mixer.candidate\_generation.RelatedTweetCandidateGenerator

import com.twitter.cr\_mixer.candidate\_generation.RelatedVideoTweetCandidateGenerator

import com.twitter.cr\_mixer.candidate\_generation.TopicTweetCandidateGenerator

import com.twitter.cr\_mixer.candidate\_generation.UtegTweetCandidateGenerator

import com.twitter.cr\_mixer.featureswitch.ParamsBuilder

import com.twitter.cr\_mixer.logging.CrMixerScribeLogger

import com.twitter.cr\_mixer.logging.RelatedTweetScribeLogger

import com.twitter.cr\_mixer.logging.AdsRecommendationsScribeLogger

import com.twitter.cr\_mixer.logging.RelatedTweetScribeMetadata

import com.twitter.cr\_mixer.logging.ScribeMetadata

import com.twitter.cr\_mixer.logging.UtegTweetScribeLogger

import com.twitter.cr\_mixer.model.AdsCandidateGeneratorQuery

import com.twitter.cr\_mixer.model.CrCandidateGeneratorQuery

import com.twitter.cr\_mixer.model.FrsTweetCandidateGeneratorQuery

import com.twitter.cr\_mixer.model.InitialCandidate

import com.twitter.cr\_mixer.model.RankedAdsCandidate

import com.twitter.cr\_mixer.model.RankedCandidate

import com.twitter.cr\_mixer.model.RelatedTweetCandidateGeneratorQuery

import com.twitter.cr\_mixer.model.RelatedVideoTweetCandidateGeneratorQuery

import com.twitter.cr\_mixer.model.TopicTweetCandidateGeneratorQuery

import com.twitter.cr\_mixer.model.TweetWithScoreAndSocialProof

import com.twitter.cr\_mixer.model.UtegTweetCandidateGeneratorQuery

import com.twitter.cr\_mixer.param.AdsParams

import com.twitter.cr\_mixer.param.FrsParams.FrsBasedCandidateGenerationMaxCandidatesNumParam

import com.twitter.cr\_mixer.param.GlobalParams

import com.twitter.cr\_mixer.param.RelatedTweetGlobalParams

import com.twitter.cr\_mixer.param.RelatedVideoTweetGlobalParams

import com.twitter.cr\_mixer.param.TopicTweetParams

import com.twitter.cr\_mixer.param.decider.CrMixerDecider

import com.twitter.cr\_mixer.param.decider.DeciderConstants

import com.twitter.cr\_mixer.param.decider.EndpointLoadShedder

import com.twitter.cr\_mixer.thriftscala.AdTweetRecommendation

import com.twitter.cr\_mixer.thriftscala.AdsRequest

import com.twitter.cr\_mixer.thriftscala.AdsResponse

import com.twitter.cr\_mixer.thriftscala.CrMixerTweetRequest

import com.twitter.cr\_mixer.thriftscala.CrMixerTweetResponse

import com.twitter.cr\_mixer.thriftscala.FrsTweetRequest

import com.twitter.cr\_mixer.thriftscala.FrsTweetResponse

import com.twitter.cr\_mixer.thriftscala.RelatedTweet

import com.twitter.cr\_mixer.thriftscala.RelatedTweetRequest

import com.twitter.cr\_mixer.thriftscala.RelatedTweetResponse

import com.twitter.cr\_mixer.thriftscala.RelatedVideoTweet

import com.twitter.cr\_mixer.thriftscala.RelatedVideoTweetRequest

import com.twitter.cr\_mixer.thriftscala.RelatedVideoTweetResponse

import com.twitter.cr\_mixer.thriftscala.TopicTweet

import com.twitter.cr\_mixer.thriftscala.TopicTweetRequest

import com.twitter.cr\_mixer.thriftscala.TopicTweetResponse

import com.twitter.cr\_mixer.thriftscala.TweetRecommendation

import com.twitter.cr\_mixer.thriftscala.UtegTweet

import com.twitter.cr\_mixer.thriftscala.UtegTweetRequest

import com.twitter.cr\_mixer.thriftscala.UtegTweetResponse

import com.twitter.cr\_mixer.util.MetricTagUtil

import com.twitter.cr\_mixer.util.SignalTimestampStatsUtil

import com.twitter.cr\_mixer.{thriftscala => t}

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finatra.thrift.Controller

import com.twitter.hermit.store.common.ReadableWritableStore

import com.twitter.simclusters\_v2.common.UserId

import com.twitter.simclusters\_v2.thriftscala.TopicId

import com.twitter.storehaus.ReadableStore

import com.twitter.timelines.timeline\_logging.{thriftscala => thriftlog}

import com.twitter.timelines.tracing.lensview.funnelseries.TweetScoreFunnelSeries

import com.twitter.util.Future

import com.twitter.util.Time

import java.util.UUID

import javax.inject.Inject

import org.apache.commons.lang.exception.ExceptionUtils

class CrMixerThriftController @Inject() (

crCandidateGenerator: CrCandidateGenerator,

relatedTweetCandidateGenerator: RelatedTweetCandidateGenerator,

relatedVideoTweetCandidateGenerator: RelatedVideoTweetCandidateGenerator,

utegTweetCandidateGenerator: UtegTweetCandidateGenerator,

frsTweetCandidateGenerator: FrsTweetCandidateGenerator,

topicTweetCandidateGenerator: TopicTweetCandidateGenerator,

crMixerScribeLogger: CrMixerScribeLogger,

relatedTweetScribeLogger: RelatedTweetScribeLogger,

utegTweetScribeLogger: UtegTweetScribeLogger,

adsRecommendationsScribeLogger: AdsRecommendationsScribeLogger,

adsCandidateGenerator: AdsCandidateGenerator,

decider: CrMixerDecider,

paramsBuilder: ParamsBuilder,

endpointLoadShedder: EndpointLoadShedder,

signalTimestampStatsUtil: SignalTimestampStatsUtil,

tweetRecommendationResultsStore: ReadableWritableStore[UserId, CrMixerTweetResponse],

userStateStore: ReadableStore[UserId, UserState],

statsReceiver: StatsReceiver)

extends Controller(t.CrMixer) {

lazy private val tweetScoreFunnelSeries = new TweetScoreFunnelSeries(statsReceiver)

private def logErrMessage(endpoint: String, e: Throwable): Unit = {

val msg = Seq(

s"Failed endpoint $endpoint: ${e.getLocalizedMessage}",

ExceptionUtils.getStackTrace(e)

).mkString("\n")

/\*\* \*

\* We chose logger.info() here to print message instead of logger.error since that

\* logger.error sometimes suppresses detailed stacktrace.

\*/

logger.info(msg)

}

private def generateRequestUUID(): Long = {

/\*\* \*

\* We generate unique UUID via bitwise operations. See the below link for more:

\* https://stackoverflow.com/questions/15184820/how-to-generate-unique-positive-long-using-uuid

\*/

UUID.randomUUID().getMostSignificantBits & Long.MaxValue

}

handle(t.CrMixer.GetTweetRecommendations) { args: t.CrMixer.GetTweetRecommendations.Args =>

val endpointName = "getTweetRecommendations"

val requestUUID = generateRequestUUID()

val startTime = Time.now.inMilliseconds

val userId = args.request.clientContext.userId.getOrElse(

throw new IllegalArgumentException("userId must be present in the Thrift clientContext")

)

val queryFut = buildCrCandidateGeneratorQuery(args.request, requestUUID, userId)

queryFut.flatMap { query =>

val scribeMetadata = ScribeMetadata.from(query)

endpointLoadShedder(endpointName, query.product.originalName) {

val response = crCandidateGenerator.get(query)

val blueVerifiedScribedResponse = response.flatMap { rankedCandidates =>

val hasBlueVerifiedCandidate = rankedCandidates.exists { tweet =>

tweet.tweetInfo.hasBlueVerifiedAnnotation.contains(true)

}

if (hasBlueVerifiedCandidate) {

crMixerScribeLogger.scribeGetTweetRecommendationsForBlueVerified(

scribeMetadata,

response)

} else {

response

}

}

val thriftResponse = blueVerifiedScribedResponse.map { candidates =>

if (query.product == t.Product.Home) {

scribeTweetScoreFunnelSeries(candidates)

}

buildThriftResponse(candidates)

}

cacheTweetRecommendationResults(args.request, thriftResponse)

crMixerScribeLogger.scribeGetTweetRecommendations(

args.request,

startTime,

scribeMetadata,

thriftResponse)

}.rescue {

case EndpointLoadShedder.LoadSheddingException =>

Future(CrMixerTweetResponse(Seq.empty))

case e =>

logErrMessage(endpointName, e)

Future(CrMixerTweetResponse(Seq.empty))

}

}

}

/\*\* \*

\* GetRelatedTweetsForQueryTweet and GetRelatedTweetsForQueryAuthor are essentially

\* doing very similar things, except that one passes in TweetId which calls TweetBased engine,

\* and the other passes in AuthorId which calls ProducerBased engine.

\*/

handle(t.CrMixer.GetRelatedTweetsForQueryTweet) {

args: t.CrMixer.GetRelatedTweetsForQueryTweet.Args =>

val endpointName = "getRelatedTweetsForQueryTweet"

getRelatedTweets(endpointName, args.request)

}

handle(t.CrMixer.GetRelatedVideoTweetsForQueryTweet) {

args: t.CrMixer.GetRelatedVideoTweetsForQueryTweet.Args =>

val endpointName = "getRelatedVideoTweetsForQueryVideoTweet"

getRelatedVideoTweets(endpointName, args.request)

}

handle(t.CrMixer.GetRelatedTweetsForQueryAuthor) {

args: t.CrMixer.GetRelatedTweetsForQueryAuthor.Args =>

val endpointName = "getRelatedTweetsForQueryAuthor"

getRelatedTweets(endpointName, args.request)

}

private def getRelatedTweets(

endpointName: String,

request: RelatedTweetRequest

): Future[RelatedTweetResponse] = {

val requestUUID = generateRequestUUID()

val startTime = Time.now.inMilliseconds

val queryFut = buildRelatedTweetQuery(request, requestUUID)

queryFut.flatMap { query =>

val relatedTweetScribeMetadata = RelatedTweetScribeMetadata.from(query)

endpointLoadShedder(endpointName, query.product.originalName) {

relatedTweetScribeLogger.scribeGetRelatedTweets(

request,

startTime,

relatedTweetScribeMetadata,

relatedTweetCandidateGenerator

.get(query)

.map(buildRelatedTweetResponse))

}.rescue {

case EndpointLoadShedder.LoadSheddingException =>

Future(RelatedTweetResponse(Seq.empty))

case e =>

logErrMessage(endpointName, e)

Future(RelatedTweetResponse(Seq.empty))

}

}

}

private def getRelatedVideoTweets(

endpointName: String,

request: RelatedVideoTweetRequest

): Future[RelatedVideoTweetResponse] = {

val requestUUID = generateRequestUUID()

val queryFut = buildRelatedVideoTweetQuery(request, requestUUID)

queryFut.flatMap { query =>

endpointLoadShedder(endpointName, query.product.originalName) {

relatedVideoTweetCandidateGenerator.get(query).map { initialCandidateSeq =>

buildRelatedVideoTweetResponse(initialCandidateSeq)

}

}.rescue {

case EndpointLoadShedder.LoadSheddingException =>

Future(RelatedVideoTweetResponse(Seq.empty))

case e =>

logErrMessage(endpointName, e)

Future(RelatedVideoTweetResponse(Seq.empty))

}

}

}

handle(t.CrMixer.GetFrsBasedTweetRecommendations) {

args: t.CrMixer.GetFrsBasedTweetRecommendations.Args =>

val endpointName = "getFrsBasedTweetRecommendations"

val requestUUID = generateRequestUUID()

val queryFut = buildFrsBasedTweetQuery(args.request, requestUUID)

queryFut.flatMap { query =>

endpointLoadShedder(endpointName, query.product.originalName) {

frsTweetCandidateGenerator.get(query).map(FrsTweetResponse(\_))

}.rescue {

case e =>

logErrMessage(endpointName, e)

Future(FrsTweetResponse(Seq.empty))

}

}

}

handle(t.CrMixer.GetTopicTweetRecommendations) {

args: t.CrMixer.GetTopicTweetRecommendations.Args =>

val endpointName = "getTopicTweetRecommendations"

val requestUUID = generateRequestUUID()

val query = buildTopicTweetQuery(args.request, requestUUID)

endpointLoadShedder(endpointName, query.product.originalName) {

topicTweetCandidateGenerator.get(query).map(TopicTweetResponse(\_))

}.rescue {

case e =>

logErrMessage(endpointName, e)

Future(TopicTweetResponse(Map.empty[Long, Seq[TopicTweet]]))

}

}

handle(t.CrMixer.GetUtegTweetRecommendations) {

args: t.CrMixer.GetUtegTweetRecommendations.Args =>

val endpointName = "getUtegTweetRecommendations"

val requestUUID = generateRequestUUID()

val startTime = Time.now.inMilliseconds

val queryFut = buildUtegTweetQuery(args.request, requestUUID)

queryFut

.flatMap { query =>

val scribeMetadata = ScribeMetadata.from(query)

endpointLoadShedder(endpointName, query.product.originalName) {

utegTweetScribeLogger.scribeGetUtegTweetRecommendations(

args.request,

startTime,

scribeMetadata,

utegTweetCandidateGenerator

.get(query)

.map(buildUtegTweetResponse)

)

}.rescue {

case e =>

logErrMessage(endpointName, e)

Future(UtegTweetResponse(Seq.empty))

}

}

}

handle(t.CrMixer.GetAdsRecommendations) { args: t.CrMixer.GetAdsRecommendations.Args =>

val endpointName = "getAdsRecommendations"

val queryFut = buildAdsCandidateGeneratorQuery(args.request)

val startTime = Time.now.inMilliseconds

queryFut.flatMap { query =>

{

val scribeMetadata = ScribeMetadata.from(query)

val response = adsCandidateGenerator

.get(query).map { candidates =>

buildAdsResponse(candidates)

}

adsRecommendationsScribeLogger.scribeGetAdsRecommendations(

args.request,

startTime,

scribeMetadata,

response,

query.params(AdsParams.EnableScribe)

)

}.rescue {

case e =>

logErrMessage(endpointName, e)

Future(AdsResponse(Seq.empty))

}

}

}

private def buildCrCandidateGeneratorQuery(

thriftRequest: CrMixerTweetRequest,

requestUUID: Long,

userId: Long

): Future[CrCandidateGeneratorQuery] = {

val product = thriftRequest.product

val productContext = thriftRequest.productContext

val scopedStats = statsReceiver

.scope(product.toString).scope("CrMixerTweetRequest")

userStateStore

.get(userId).map { userStateOpt =>

val userState = userStateOpt

.getOrElse(UserState.EnumUnknownUserState(100))

scopedStats.scope("UserState").counter(userState.toString).incr()

val params =

paramsBuilder.buildFromClientContext(

thriftRequest.clientContext,

thriftRequest.product,

userState

)

// Specify product-specific behavior mapping here

val maxNumResults = (product, productContext) match {

case (t.Product.Home, Some(t.ProductContext.HomeContext(homeContext))) =>

homeContext.maxResults.getOrElse(9999)

case (t.Product.Notifications, Some(t.ProductContext.NotificationsContext(cxt))) =>

params(GlobalParams.MaxCandidatesPerRequestParam)

case (t.Product.Email, None) =>

params(GlobalParams.MaxCandidatesPerRequestParam)

case (t.Product.ImmersiveMediaViewer, None) =>

params(GlobalParams.MaxCandidatesPerRequestParam)

case (t.Product.VideoCarousel, None) =>

params(GlobalParams.MaxCandidatesPerRequestParam)

case \_ =>

throw new IllegalArgumentException(

s"Product ${product} and ProductContext ${productContext} are not allowed in CrMixer"

)

}

CrCandidateGeneratorQuery(

userId = userId,

product = product,

userState = userState,

maxNumResults = maxNumResults,

impressedTweetList = thriftRequest.excludedTweetIds.getOrElse(Nil).toSet,

params = params,

requestUUID = requestUUID,

languageCode = thriftRequest.clientContext.languageCode

)

}

}

private def buildRelatedTweetQuery(

thriftRequest: RelatedTweetRequest,

requestUUID: Long

): Future[RelatedTweetCandidateGeneratorQuery] = {

val product = thriftRequest.product

val scopedStats = statsReceiver

.scope(product.toString).scope("RelatedTweetRequest")

val userStateFut: Future[UserState] = (thriftRequest.clientContext.userId match {

case Some(userId) => userStateStore.get(userId)

case None => Future.value(Some(UserState.EnumUnknownUserState(100)))

}).map(\_.getOrElse(UserState.EnumUnknownUserState(100)))

userStateFut.map { userState =>

scopedStats.scope("UserState").counter(userState.toString).incr()

val params =

paramsBuilder.buildFromClientContext(

thriftRequest.clientContext,

thriftRequest.product,

userState)

// Specify product-specific behavior mapping here

// Currently, Home takes 10, and RUX takes 100

val maxNumResults = params(RelatedTweetGlobalParams.MaxCandidatesPerRequestParam)

RelatedTweetCandidateGeneratorQuery(

internalId = thriftRequest.internalId,

clientContext = thriftRequest.clientContext,

product = product,

maxNumResults = maxNumResults,

impressedTweetList = thriftRequest.excludedTweetIds.getOrElse(Nil).toSet,

params = params,

requestUUID = requestUUID

)

}

}

private def buildAdsCandidateGeneratorQuery(

thriftRequest: AdsRequest

): Future[AdsCandidateGeneratorQuery] = {

val userId = thriftRequest.clientContext.userId.getOrElse(

throw new IllegalArgumentException("userId must be present in the Thrift clientContext")

)

val product = thriftRequest.product

val requestUUID = generateRequestUUID()

userStateStore

.get(userId).map { userStateOpt =>

val userState = userStateOpt

.getOrElse(UserState.EnumUnknownUserState(100))

val params =

paramsBuilder.buildFromClientContext(

thriftRequest.clientContext,

thriftRequest.product,

userState)

val maxNumResults = params(AdsParams.AdsCandidateGenerationMaxCandidatesNumParam)

AdsCandidateGeneratorQuery(

userId = userId,

product = product,

userState = userState,

params = params,

maxNumResults = maxNumResults,

requestUUID = requestUUID

)

}

}

private def buildRelatedVideoTweetQuery(

thriftRequest: RelatedVideoTweetRequest,

requestUUID: Long

): Future[RelatedVideoTweetCandidateGeneratorQuery] = {

val product = thriftRequest.product

val scopedStats = statsReceiver

.scope(product.toString).scope("RelatedVideoTweetRequest")

val userStateFut: Future[UserState] = (thriftRequest.clientContext.userId match {

case Some(userId) => userStateStore.get(userId)

case None => Future.value(Some(UserState.EnumUnknownUserState(100)))

}).map(\_.getOrElse(UserState.EnumUnknownUserState(100)))

userStateFut.map { userState =>

scopedStats.scope("UserState").counter(userState.toString).incr()

val params =

paramsBuilder.buildFromClientContext(

thriftRequest.clientContext,

thriftRequest.product,

userState)

val maxNumResults = params(RelatedVideoTweetGlobalParams.MaxCandidatesPerRequestParam)

RelatedVideoTweetCandidateGeneratorQuery(

internalId = thriftRequest.internalId,

clientContext = thriftRequest.clientContext,

product = product,

maxNumResults = maxNumResults,

impressedTweetList = thriftRequest.excludedTweetIds.getOrElse(Nil).toSet,

params = params,

requestUUID = requestUUID

)

}

}

private def buildUtegTweetQuery(

thriftRequest: UtegTweetRequest,

requestUUID: Long

): Future[UtegTweetCandidateGeneratorQuery] = {

val userId = thriftRequest.clientContext.userId.getOrElse(

throw new IllegalArgumentException("userId must be present in the Thrift clientContext")

)

val product = thriftRequest.product

val productContext = thriftRequest.productContext

val scopedStats = statsReceiver

.scope(product.toString).scope("UtegTweetRequest")

userStateStore

.get(userId).map { userStateOpt =>

val userState = userStateOpt

.getOrElse(UserState.EnumUnknownUserState(100))

scopedStats.scope("UserState").counter(userState.toString).incr()

val params =

paramsBuilder.buildFromClientContext(

thriftRequest.clientContext,

thriftRequest.product,

userState

)

// Specify product-specific behavior mapping here

val maxNumResults = (product, productContext) match {

case (t.Product.Home, Some(t.ProductContext.HomeContext(homeContext))) =>

homeContext.maxResults.getOrElse(9999)

case \_ =>

throw new IllegalArgumentException(

s"Product ${product} and ProductContext ${productContext} are not allowed in CrMixer"

)

}

UtegTweetCandidateGeneratorQuery(

userId = userId,

product = product,

userState = userState,

maxNumResults = maxNumResults,

impressedTweetList = thriftRequest.excludedTweetIds.getOrElse(Nil).toSet,

params = params,

requestUUID = requestUUID

)

}

}

private def buildTopicTweetQuery(

thriftRequest: TopicTweetRequest,

requestUUID: Long

): TopicTweetCandidateGeneratorQuery = {

val userId = thriftRequest.clientContext.userId.getOrElse(

throw new IllegalArgumentException(

"userId must be present in the TopicTweetRequest clientContext")

)

val product = thriftRequest.product

val productContext = thriftRequest.productContext

// Specify product-specific behavior mapping here

val isVideoOnly = (product, productContext) match {

case (t.Product.ExploreTopics, Some(t.ProductContext.ExploreContext(context))) =>

context.isVideoOnly

case (t.Product.TopicLandingPage, None) =>

false

case (t.Product.HomeTopicsBackfill, None) =>

false

case (t.Product.TopicTweetsStrato, None) =>

false

case \_ =>

throw new IllegalArgumentException(

s"Product ${product} and ProductContext ${productContext} are not allowed in CrMixer"

)

}

statsReceiver.scope(product.toString).counter(TopicTweetRequest.toString).incr()

val params =

paramsBuilder.buildFromClientContext(

thriftRequest.clientContext,

product,

UserState.EnumUnknownUserState(100)

)

val topicIds = thriftRequest.topicIds.map { topicId =>

TopicId(

entityId = topicId,

language = thriftRequest.clientContext.languageCode,

country = None

)

}.toSet

TopicTweetCandidateGeneratorQuery(

userId = userId,

topicIds = topicIds,

product = product,

maxNumResults = params(TopicTweetParams.MaxTopicTweetCandidatesParam),

impressedTweetList = thriftRequest.excludedTweetIds.getOrElse(Nil).toSet,

params = params,

requestUUID = requestUUID,

isVideoOnly = isVideoOnly

)

}

private def buildFrsBasedTweetQuery(

thriftRequest: FrsTweetRequest,

requestUUID: Long

): Future[FrsTweetCandidateGeneratorQuery] = {

val userId = thriftRequest.clientContext.userId.getOrElse(

throw new IllegalArgumentException(

"userId must be present in the FrsTweetRequest clientContext")

)

val product = thriftRequest.product

val productContext = thriftRequest.productContext

val scopedStats = statsReceiver

.scope(product.toString).scope("FrsTweetRequest")

userStateStore

.get(userId).map { userStateOpt =>

val userState = userStateOpt

.getOrElse(UserState.EnumUnknownUserState(100))

scopedStats.scope("UserState").counter(userState.toString).incr()

val params =

paramsBuilder.buildFromClientContext(

thriftRequest.clientContext,

thriftRequest.product,

userState

)

val maxNumResults = (product, productContext) match {

case (t.Product.Home, Some(t.ProductContext.HomeContext(homeContext))) =>

homeContext.maxResults.getOrElse(

params(FrsBasedCandidateGenerationMaxCandidatesNumParam))

case \_ =>

params(FrsBasedCandidateGenerationMaxCandidatesNumParam)

}

FrsTweetCandidateGeneratorQuery(

userId = userId,

product = product,

maxNumResults = maxNumResults,

impressedTweetList = thriftRequest.excludedTweetIds.getOrElse(Nil).toSet,

impressedUserList = thriftRequest.excludedUserIds.getOrElse(Nil).toSet,

params = params,

languageCodeOpt = thriftRequest.clientContext.languageCode,

countryCodeOpt = thriftRequest.clientContext.countryCode,

requestUUID = requestUUID

)

}

}

private def buildThriftResponse(

candidates: Seq[RankedCandidate]

): CrMixerTweetResponse = {

val tweets = candidates.map { candidate =>

TweetRecommendation(

tweetId = candidate.tweetId,

score = candidate.predictionScore,

metricTags = Some(MetricTagUtil.buildMetricTags(candidate)),

latestSourceSignalTimestampInMillis =

SignalTimestampStatsUtil.buildLatestSourceSignalTimestamp(candidate)

)

}

signalTimestampStatsUtil.statsSignalTimestamp(tweets)

CrMixerTweetResponse(tweets)

}

private def scribeTweetScoreFunnelSeries(

candidates: Seq[RankedCandidate]

): Seq[RankedCandidate] = {

// 202210210901 is a random number for code search of Lensview

tweetScoreFunnelSeries.startNewSpan(

name = "GetTweetRecommendationsTopLevelTweetSimilarityEngineType",

codePtr = 202210210901L) {

(

candidates,

candidates.map { candidate =>

thriftlog.TweetDimensionMeasure(

dimension = Some(

thriftlog

.RequestTweetDimension(

candidate.tweetId,

candidate.reasonChosen.similarityEngineInfo.similarityEngineType.value)),

measure = Some(thriftlog.RequestTweetMeasure(candidate.predictionScore))

)

}

)

}

}

private def buildRelatedTweetResponse(candidates: Seq[InitialCandidate]): RelatedTweetResponse = {

val tweets = candidates.map { candidate =>

RelatedTweet(

tweetId = candidate.tweetId,

score = Some(candidate.getSimilarityScore),

authorId = Some(candidate.tweetInfo.authorId)

)

}

RelatedTweetResponse(tweets)

}

private def buildRelatedVideoTweetResponse(

candidates: Seq[InitialCandidate]

): RelatedVideoTweetResponse = {

val tweets = candidates.map { candidate =>

RelatedVideoTweet(

tweetId = candidate.tweetId,

score = Some(candidate.getSimilarityScore)

)

}

RelatedVideoTweetResponse(tweets)

}

private def buildUtegTweetResponse(

candidates: Seq[TweetWithScoreAndSocialProof]

): UtegTweetResponse = {

val tweets = candidates.map { candidate =>

UtegTweet(

tweetId = candidate.tweetId,

score = candidate.score,

socialProofByType = candidate.socialProofByType

)

}

UtegTweetResponse(tweets)

}

private def buildAdsResponse(

candidates: Seq[RankedAdsCandidate]

): AdsResponse = {

AdsResponse(ads = candidates.map { candidate =>

AdTweetRecommendation(

tweetId = candidate.tweetId,

score = candidate.predictionScore,

lineItems = Some(candidate.lineItemInfo))

})

}

private def cacheTweetRecommendationResults(

request: CrMixerTweetRequest,

response: Future[CrMixerTweetResponse]

): Unit = {

val userId = request.clientContext.userId.getOrElse(

throw new IllegalArgumentException(

"userId must be present in getTweetRecommendations() Thrift clientContext"))

if (decider.isAvailableForId(userId, DeciderConstants.getTweetRecommendationsCacheRate)) {

response.map { crMixerTweetResponse =>

{

(

request.product,

request.clientContext.userId,

crMixerTweetResponse.tweets.nonEmpty) match {

case (t.Product.Home, Some(userId), true) =>

tweetRecommendationResultsStore.put((userId, crMixerTweetResponse))

case \_ => Future.value(Unit)

}

}

}

}

}

}