package com.twitter.frigate.pushservice.send\_handler

import com.twitter.finagle.stats.BroadcastStatsReceiver

import com.twitter.finagle.stats.Stat

import com.twitter.finagle.stats.StatsReceiver

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

import com.twitter.frigate.common.base.CandidateFilteringOnlyFlow

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

import com.twitter.frigate.common.base.FeatureMap

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

import com.twitter.frigate.common.base.Response

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

import com.twitter.frigate.common.base.Stats.track

import com.twitter.frigate.common.config.CommonConstants

import com.twitter.frigate.common.logger.MRLogger

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

import com.twitter.frigate.common.util.InvalidRequestException

import com.twitter.frigate.common.util.MrNtabCopyObjects

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

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

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

import com.twitter.frigate.pushservice.config.Config

import com.twitter.frigate.pushservice.ml.HydrationContextBuilder

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

import com.twitter.frigate.pushservice.scriber.MrRequestScribeHandler

import com.twitter.frigate.pushservice.send\_handler.generator.PushRequestToCandidate

import com.twitter.frigate.pushservice.take.SendHandlerNotifier

import com.twitter.frigate.pushservice.take.candidate\_validator.SendHandlerPostCandidateValidator

import com.twitter.frigate.pushservice.take.candidate\_validator.SendHandlerPreCandidateValidator

import com.twitter.frigate.pushservice.target.PushTargetUserBuilder

import com.twitter.frigate.pushservice.util.ResponseStatsTrackUtils.trackStatsForResponseToRequest

import com.twitter.frigate.pushservice.util.SendHandlerPredicateUtil

import com.twitter.frigate.pushservice.thriftscala.PushRequest

import com.twitter.frigate.pushservice.thriftscala.PushRequestScribe

import com.twitter.frigate.pushservice.thriftscala.PushResponse

import com.twitter.frigate.thriftscala.CommonRecommendationType

import com.twitter.nrel.heavyranker.FeatureHydrator

import com.twitter.util.\_

/\*\*

\* A handler for sending PushRequests

\*/

class SendHandler(

pushTargetUserBuilder: PushTargetUserBuilder,

preCandidateValidator: SendHandlerPreCandidateValidator,

postCandidateValidator: SendHandlerPostCandidateValidator,

sendHandlerNotifier: SendHandlerNotifier,

candidateHydrator: SendHandlerPushCandidateHydrator,

featureHydrator: FeatureHydrator,

sendHandlerPredicateUtil: SendHandlerPredicateUtil,

mrRequestScriberNode: String

)(

implicit val statsReceiver: StatsReceiver,

implicit val config: Config)

extends CandidateFilteringOnlyFlow[Target, RawCandidate, PushCandidate] {

implicit private val timer: Timer = new JavaTimer(true)

val stats = statsReceiver.scope("SendHandler")

val log = MRLogger("SendHandler")

private val buildTargetStats = stats.scope("build\_target")

private val candidateHydrationLatency: Stat =

stats.stat("candidateHydrationLatency")

private val candidatePreValidatorLatency: Stat =

stats.stat("candidatePreValidatorLatency")

private val candidatePostValidatorLatency: Stat =

stats.stat("candidatePostValidatorLatency")

private val featureHydrationLatency: StatsReceiver =

stats.scope("featureHydrationLatency")

private val mrRequestScribeHandler =

new MrRequestScribeHandler(mrRequestScriberNode, stats.scope("mr\_request\_scribe"))

def apply(request: PushRequest): Future[PushResponse] = {

val receivers = Seq(

stats,

stats.scope(request.notification.commonRecommendationType.toString)

)

val bStats = BroadcastStatsReceiver(receivers)

bStats.counter("requests").incr()

Stat

.timeFuture(bStats.stat("latency"))(

process(request).raiseWithin(CommonConstants.maxPushRequestDuration))

.onSuccess {

case (pushResp, rawCandidate) =>

trackStatsForResponseToRequest(

rawCandidate.commonRecType,

rawCandidate.target,

pushResp,

receivers)(statsReceiver)

if (!request.context.exists(\_.darkWrite.contains(true))) {

config.requestScribe(PushRequestScribe(request, pushResp))

}

}

.onFailure { ex =>

bStats.counter("failures").incr()

bStats.scope("failures").counter(ex.getClass.getCanonicalName).incr()

}

.map {

case (pushResp, \_) => pushResp

}

}

private def process(request: PushRequest): Future[(PushResponse, RawCandidate)] = {

val recType = request.notification.commonRecommendationType

track(buildTargetStats)(

pushTargetUserBuilder

.buildTarget(

request.userId,

request.context

)

).flatMap { targetUser =>

val responseWithScribedInfo = request.context.exists { context =>

context.responseWithScribedInfo.contains(true)

}

val newRequest =

if (request.notification.commonRecommendationType == CommonRecommendationType.MagicFanoutNewsEvent &&

targetUser.params(EnableMagicFanoutNewsForYouNtabCopy)) {

val newNotification = request.notification.copy(ntabCopyId =

Some(MrNtabCopyObjects.MagicFanoutNewsForYouCopy.copyId))

request.copy(notification = newNotification)

} else request

if (RecTypes.isSendHandlerType(recType) || newRequest.context.exists(

\_.allowCRT.contains(true))) {

val rawCandidateFut = PushRequestToCandidate.generatePushCandidate(

newRequest.notification,

targetUser

)

rawCandidateFut.flatMap { rawCandidate =>

val pushResponse = process(targetUser, Seq(rawCandidate)).flatMap {

sendHandlerNotifier.checkResponseAndNotify(\_, responseWithScribedInfo)

}

pushResponse.map { pushResponse =>

(pushResponse, rawCandidate)

}

}

} else {

Future.exception(InvalidRequestException(s"${recType.name} not supported in SendHandler"))

}

}

}

private def hydrateFeatures(

candidateDetails: Seq[CandidateDetails[PushCandidate]],

target: Target,

): Future[Seq[CandidateDetails[PushCandidate]]] = {

candidateDetails.headOption match {

case Some(candidateDetail)

if RecTypes.notEligibleForModelScoreTracking(candidateDetail.candidate.commonRecType) =>

Future.value(candidateDetails)

case Some(candidateDetail) =>

val hydrationContextFut = HydrationContextBuilder.build(candidateDetail.candidate)

hydrationContextFut.flatMap { hc =>

featureHydrator

.hydrateCandidate(Seq(hc), target.mrRequestContextForFeatureStore)

.map { hydrationResult =>

val features = hydrationResult.getOrElse(hc, FeatureMap())

candidateDetail.candidate.mergeFeatures(features)

candidateDetails

}

}

case \_ => Future.Nil

}

}

override def process(

target: Target,

externalCandidates: Seq[RawCandidate]

): Future[Response[PushCandidate, Result]] = {

val candidate = externalCandidates.map(CandidateDetails(\_, "realtime"))

for {

hydratedCandidatesWithCopy <- hydrateCandidates(candidate)

(candidates, preHydrationFilteredCandidates) <- track(filterStats)(

filter(target, hydratedCandidatesWithCopy)

)

featureHydratedCandidates <-

track(featureHydrationLatency)(hydrateFeatures(candidates, target))

allTakeCandidateResults <- track(takeStats)(

take(target, featureHydratedCandidates, desiredCandidateCount(target))

)

\_ <- mrRequestScribeHandler.scribeForCandidateFiltering(

target = target,

hydratedCandidates = hydratedCandidatesWithCopy,

preRankingFilteredCandidates = preHydrationFilteredCandidates,

rankedCandidates = featureHydratedCandidates,

rerankedCandidates = Seq.empty,

restrictFilteredCandidates = Seq.empty, // no restrict step

allTakeCandidateResults = allTakeCandidateResults

)

} yield {

/\*\*

\* We combine the results for all filtering steps and pass on in sequence to next step

\*

\* This is done to ensure the filtering reason for the candidate from multiple levels of

\* filtering is carried all the way until [[PushResponse]] is built and returned from

\* frigate-pushservice-send

\*/

Response(OK, allTakeCandidateResults ++ preHydrationFilteredCandidates)

}

}

override def hydrateCandidates(

candidates: Seq[CandidateDetails[RawCandidate]]

): Future[Seq[CandidateDetails[PushCandidate]]] = {

Stat.timeFuture(candidateHydrationLatency)(candidateHydrator(candidates))

}

// Filter Step - pre-predicates and app specific predicates

override def filter(

target: Target,

hydratedCandidatesDetails: Seq[CandidateDetails[PushCandidate]]

): Future[

(Seq[CandidateDetails[PushCandidate]], Seq[CandidateResult[PushCandidate, Result]])

] = {

Stat.timeFuture(candidatePreValidatorLatency)(

sendHandlerPredicateUtil.preValidationForCandidate(

hydratedCandidatesDetails,

preCandidateValidator

))

}

// Post Validation - Take step

override def validCandidates(

target: Target,

candidates: Seq[PushCandidate]

): Future[Seq[Result]] = {

Stat.timeFuture(candidatePostValidatorLatency)(Future.collect(candidates.map { candidate =>

sendHandlerPredicateUtil

.postValidationForCandidate(candidate, postCandidateValidator)

.map(res => res.result)

}))

}

}