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

import com.twitter.finagle.stats.Counter

import com.twitter.finagle.stats.StatsReceiver

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

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

import com.twitter.frigate.common.base.CandidateSource

import com.twitter.frigate.common.base.FetchRankFlowWithHydratedCandidates

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

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.base.Stats.trackSeq

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

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.adaptor.LoggedOutPushCandidateSourceGenerator

import com.twitter.frigate.pushservice.predicate.LoggedOutPreRankingPredicates

import com.twitter.frigate.pushservice.predicate.LoggedOutTargetPredicates

import com.twitter.frigate.pushservice.rank.LoggedOutRanker

import com.twitter.frigate.pushservice.take.LoggedOutRefreshForPushNotifier

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

import com.twitter.frigate.pushservice.target.LoggedOutPushTargetUserBuilder

import com.twitter.frigate.pushservice.thriftscala.LoggedOutRequest

import com.twitter.frigate.pushservice.thriftscala.LoggedOutResponse

import com.twitter.frigate.pushservice.thriftscala.PushContext

import com.twitter.hermit.predicate.NamedPredicate

import com.twitter.hermit.predicate.Predicate

import com.twitter.hermit.predicate.SequentialPredicate

import com.twitter.util.Future

class LoggedOutRefreshForPushHandler(

val loPushTargetUserBuilder: LoggedOutPushTargetUserBuilder,

val loPushCandidateSourceGenerator: LoggedOutPushCandidateSourceGenerator,

candidateHydrator: PushCandidateHydrator,

val loRanker: LoggedOutRanker,

val loRfphNotifier: LoggedOutRefreshForPushNotifier,

loMrRequestScriberNode: String

)(

globalStats: StatsReceiver)

extends FetchRankFlowWithHydratedCandidates[Target, RawCandidate, PushCandidate] {

val log = MRLogger("LORefreshForPushHandler")

implicit val statsReceiver: StatsReceiver =

globalStats.scope("LORefreshForPushHandler")

private val loggedOutBuildStats = statsReceiver.scope("logged\_out\_build\_target")

private val loggedOutProcessStats = statsReceiver.scope("logged\_out\_process")

private val loggedOutNotifyStats = statsReceiver.scope("logged\_out\_notify")

private val loCandidateHydrationStats: StatsReceiver =

statsReceiver.scope("logged\_out\_candidate\_hydration")

val mrLORequestCandidateScribeStats =

statsReceiver.scope("mr\_logged\_out\_request\_scribe\_candidates")

val mrRequestScribeHandler =

new MrRequestScribeHandler(loMrRequestScriberNode, statsReceiver.scope("lo\_mr\_request\_scribe"))

val loMrRequestTargetScribeStats = statsReceiver.scope("lo\_mr\_request\_scribe\_target")

lazy val loCandSourceEligibleCounter: Counter =

loCandidateStats.counter("logged\_out\_cand\_source\_eligible")

lazy val loCandSourceNotEligibleCounter: Counter =

loCandidateStats.counter("logged\_out\_cand\_source\_not\_eligible")

lazy val allCandidatesCounter: Counter = statsReceiver.counter("all\_logged\_out\_candidates")

val allCandidatesFilteredPreRank = filterStats.counter("all\_logged\_out\_candidates\_filtered")

override def targetPredicates(target: Target): List[Predicate[Target]] = List(

LoggedOutTargetPredicates.targetFatiguePredicate(),

LoggedOutTargetPredicates.loggedOutRecsHoldbackPredicate()

)

override def isTargetValid(target: Target): Future[Result] = {

val resultFut =

if (target.skipFilters) {

Future.value(OK)

} else {

predicateSeq(target).track(Seq(target)).map { resultArr =>

trackTargetPredStats(resultArr(0))

}

}

track(targetStats)(resultFut)

}

override def rank(

target: Target,

candidateDetails: Seq[CandidateDetails[PushCandidate]]

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

loRanker.rank(candidateDetails)

}

override def validCandidates(

target: Target,

candidates: Seq[PushCandidate]

): Future[Seq[Result]] = {

Future.value(candidates.map { c => OK })

}

override def desiredCandidateCount(target: Target): Int = 1

private val loggedOutPreRankingPredicates =

LoggedOutPreRankingPredicates(filterStats.scope("logged\_out\_predicates"))

private val loggedOutPreRankingPredicateChain =

new SequentialPredicate[PushCandidate](loggedOutPreRankingPredicates)

override def filter(

target: Target,

candidates: Seq[CandidateDetails[PushCandidate]]

): Future[

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

] = {

val predicateChain = loggedOutPreRankingPredicateChain

predicateChain

.track(candidates.map(\_.candidate))

.map { results =>

val resultForPreRankingFiltering =

results

.zip(candidates)

.foldLeft(

(

Seq.empty[CandidateDetails[PushCandidate]],

Seq.empty[CandidateResult[PushCandidate, Result]]

)

) {

case ((goodCandidates, filteredCandidates), (result, candidateDetails)) =>

result match {

case None =>

(goodCandidates :+ candidateDetails, filteredCandidates)

case Some(pred: NamedPredicate[\_]) =>

val r = Invalid(Some(pred.name))

(

goodCandidates,

filteredCandidates :+ CandidateResult[PushCandidate, Result](

candidateDetails.candidate,

candidateDetails.source,

r

)

)

case Some(\_) =>

val r = Invalid(Some("Filtered by un-named predicate"))

(

goodCandidates,

filteredCandidates :+ CandidateResult[PushCandidate, Result](

candidateDetails.candidate,

candidateDetails.source,

r

)

)

}

}

resultForPreRankingFiltering match {

case (validCandidates, \_) if validCandidates.isEmpty && candidates.nonEmpty =>

allCandidatesFilteredPreRank.incr()

case \_ => ()

}

resultForPreRankingFiltering

}

}

override def candidateSources(

target: Target

): Future[Seq[CandidateSource[Target, RawCandidate]]] = {

Future

.collect(loPushCandidateSourceGenerator.sources.map { cs =>

cs.isCandidateSourceAvailable(target).map { isEligible =>

if (isEligible) {

loCandSourceEligibleCounter.incr()

Some(cs)

} else {

loCandSourceNotEligibleCounter.incr()

None

}

}

}).map(\_.flatten)

}

override def process(

target: Target,

externalCandidates: Seq[RawCandidate] = Nil

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

isTargetValid(target).flatMap {

case OK =>

for {

candidatesFromSources <- trackSeq(fetchStats)(fetchCandidates(target))

externalCandidateDetails = externalCandidates.map(

CandidateDetails(\_, "logged\_out\_refresh\_for\_push\_handler\_external\_candidates"))

allCandidates = candidatesFromSources ++ externalCandidateDetails

hydratedCandidatesWithCopy <-

trackSeq(loCandidateHydrationStats)(hydrateCandidates(allCandidates))

(candidates, preRankingFilteredCandidates) <-

track(filterStats)(filter(target, hydratedCandidatesWithCopy))

rankedCandidates <- trackSeq(rankingStats)(rank(target, candidates))

allTakeCandidateResults <- track(takeStats)(

take(target, rankedCandidates, desiredCandidateCount(target))

)

\_ <- track(mrLORequestCandidateScribeStats)(

mrRequestScribeHandler.scribeForCandidateFiltering(

target,

hydratedCandidatesWithCopy,

preRankingFilteredCandidates,

rankedCandidates,

rankedCandidates,

rankedCandidates,

allTakeCandidateResults

))

} yield {

val takeCandidateResults = allTakeCandidateResults.filterNot { candResult =>

candResult.result == MoreThanDesiredCandidates

}

val allCandidateResults = takeCandidateResults ++ preRankingFilteredCandidates

allCandidatesCounter.incr(allCandidateResults.size)

Response(OK, allCandidateResults)

}

case result: Result =>

for (\_ <- track(loMrRequestTargetScribeStats)(

mrRequestScribeHandler.scribeForTargetFiltering(target, result))) yield {

Response(result, Nil)

}

}

}

def buildTarget(

guestId: Long,

inputPushContext: Option[PushContext]

): Future[Target] =

loPushTargetUserBuilder.buildTarget(guestId, inputPushContext)

/\*\*

\* Hydrate candidate by querying downstream services

\*

\* @param candidates - candidates

\*

\* @return - hydrated candidates

\*/

override def hydrateCandidates(

candidates: Seq[CandidateDetails[RawCandidate]]

): Future[Seq[CandidateDetails[PushCandidate]]] = candidateHydrator(candidates)

override def batchForCandidatesCheck(target: Target): Int = 1

def refreshAndSend(request: LoggedOutRequest): Future[LoggedOutResponse] = {

for {

target <- track(loggedOutBuildStats)(

loPushTargetUserBuilder.buildTarget(request.guestId, request.context))

response <- track(loggedOutProcessStats)(process(target, externalCandidates = Seq.empty))

loggedOutRefreshResponse <-

track(loggedOutNotifyStats)(loRfphNotifier.checkResponseAndNotify(response))

} yield {

loggedOutRefreshResponse

}

}

}