package com.twitter.frigate.pushservice.take.candidate\_validator

import com.twitter.finagle.stats.StatsReceiver

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

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

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

import com.twitter.frigate.pushservice.take.predicates.TakeCommonPredicates

import com.twitter.frigate.thriftscala.CommonRecommendationType

import com.twitter.hermit.predicate.ConcurrentPredicate

import com.twitter.hermit.predicate.NamedPredicate

import com.twitter.hermit.predicate.Predicate

import com.twitter.hermit.predicate.SequentialPredicate

import com.twitter.util.Future

trait CandidateValidator extends TakeCommonPredicates {

override implicit val statsReceiver: StatsReceiver = config.statsReceiver

protected val log = MRLogger("CandidateValidator")

private lazy val skipFiltersCounter = statsReceiver.counter("enable\_skip\_filters")

private lazy val emailUserSkipFiltersCounter =

statsReceiver.counter("email\_user\_enable\_skip\_filters")

private lazy val enablePredicatesCounter = statsReceiver.counter("enable\_predicates")

protected def enabledPredicates[C <: PushCandidate](

candidate: C,

predicates: List[NamedPredicate[C]]

): List[NamedPredicate[C]] = {

val target = candidate.target

val skipFilters: Boolean =

target.pushContext.flatMap(\_.skipFilters).getOrElse(false) || target.params(

PushFeatureSwitchParams.SkipPostRankingFilters)

if (skipFilters) {

skipFiltersCounter.incr()

if (target.isEmailUser) emailUserSkipFiltersCounter.incr()

val predicatesToEnable = target.pushContext.flatMap(\_.predicatesToEnable).getOrElse(Nil)

if (predicatesToEnable.nonEmpty) enablePredicatesCounter.incr()

// if we skip predicates on pushContext, only enable the explicitly specified predicates

predicates.filter(predicatesToEnable.contains)

} else predicates

}

protected def executeSequentialPredicates[C <: PushCandidate](

candidate: C,

predicates: List[NamedPredicate[C]]

): Future[Option[Predicate[C]]] = {

val predicatesEnabled = enabledPredicates(candidate, predicates)

val sequentialPredicate = new SequentialPredicate(predicatesEnabled)

sequentialPredicate.track(Seq(candidate)).map(\_.head)

}

protected def executeConcurrentPredicates[C <: PushCandidate](

candidate: C,

predicates: List[NamedPredicate[C]]

): Future[List[Predicate[C]]] = {

val predicatesEnabled = enabledPredicates(candidate, predicates)

val concurrentPredicate: ConcurrentPredicate[C] = new ConcurrentPredicate[C](predicatesEnabled)

concurrentPredicate.track(Seq(candidate)).map(\_.head)

}

protected val candidatePredicatesMap: Map[CommonRecommendationType, List[

NamedPredicate[\_ <: PushCandidate]

]]

protected def getCRTPredicates[C <: PushCandidate](

CRT: CommonRecommendationType

): List[NamedPredicate[C]] = {

candidatePredicatesMap.get(CRT) match {

case Some(predicates) =>

predicates.asInstanceOf[List[NamedPredicate[C]]]

case \_ =>

throw new IllegalStateException(

s"Unknown CommonRecommendationType for Predicates: ${CRT.name}")

}

}

def validateCandidate[C <: PushCandidate](candidate: C): Future[Option[Predicate[C]]]

}