package com.twitter.frigate.pushservice.predicate

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.TargetUser

import com.twitter.frigate.common.candidate.FrigateHistory

import com.twitter.frigate.common.candidate.HTLVisitHistory

import com.twitter.frigate.common.candidate.TargetABDecider

import com.twitter.frigate.common.candidate.UserDetails

import com.twitter.frigate.common.predicate.TargetUserPredicates

import com.twitter.frigate.common.predicate.{FatiguePredicate => CommonFatiguePredicate}

import com.twitter.frigate.common.store.deviceinfo.MobileClientType

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

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

import com.twitter.frigate.pushservice.target.TargetScoringDetails

import com.twitter.frigate.pushservice.util.PushCapUtil

import com.twitter.frigate.thriftscala.NotificationDisplayLocation

import com.twitter.frigate.thriftscala.{CommonRecommendationType => CRT}

import com.twitter.hermit.predicate.NamedPredicate

import com.twitter.hermit.predicate.Predicate

import com.twitter.timelines.configapi.FSBoundedParam

import com.twitter.timelines.configapi.Param

import com.twitter.util.Duration

import com.twitter.util.Future

object TargetPredicates {

def paramPredicate[T <: Target](

param: Param[Boolean]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[T] = {

val name = param.getClass.getSimpleName.stripSuffix("$")

Predicate

.from { target: T => target.params(param) }

.withStats(statsReceiver.scope(s"param\_${name}\_controlled\_predicate"))

.withName(s"param\_${name}\_controlled\_predicate")

}

/\*\*

\* Use the predicate except fn is true., Same as the candidate version but for Target

\*/

def exceptedPredicate[T <: TargetUser](

name: String,

fn: T => Future[Boolean],

predicate: Predicate[T]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[T] = {

Predicate

.fromAsync { e: T => fn(e) }

.or(predicate)

.withStats(statsReceiver.scope(name))

.withName(name)

}

/\*\*

\* Refresh For push handler target user predicate to fatigue on visiting Home timeline

\*/

def targetHTLVisitPredicate[

T <: TargetUser with UserDetails with TargetABDecider with HTLVisitHistory

](

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[T] = {

val name = "target\_htl\_visit\_predicate"

Predicate

.fromAsync { target: T =>

val hoursToFatigue = target.params(PushFeatureSwitchParams.HTLVisitFatigueTime)

TargetUserPredicates

.homeTimelineFatigue(hoursToFatigue.hours)

.apply(Seq(target))

.map(\_.head)

}

.withStats(statsReceiver.scope(name))

.withName(name)

}

def targetPushBitEnabledPredicate[T <: Target](

implicit statsReceiver: StatsReceiver

): NamedPredicate[T] = {

val name = "push\_bit\_enabled"

val scopedStats = statsReceiver.scope(s"targetpredicate\_$name")

Predicate

.fromAsync { target: T =>

target.deviceInfo

.map { info =>

info.exists { deviceInfo =>

deviceInfo.isRecommendationsEligible ||

deviceInfo.isNewsEligible ||

deviceInfo.isTopicsEligible ||

deviceInfo.isSpacesEligible

}

}

}.withStats(scopedStats)

.withName(name)

}

def targetFatiguePredicate[T <: Target](

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[T] = {

val name = "target\_fatigue\_predicate"

val predicateStatScope = statsReceiver.scope(name)

Predicate

.fromAsync { target: T =>

PushCapUtil

.getPushCapFatigue(target, predicateStatScope)

.flatMap { pushCapInfo =>

CommonFatiguePredicate

.magicRecsPushTargetFatiguePredicate(

interval = pushCapInfo.fatigueInterval,

maxInInterval = pushCapInfo.pushcap

)

.apply(Seq(target))

.map(\_.headOption.getOrElse(false))

}

}

.withStats(predicateStatScope)

.withName(name)

}

def teamExceptedPredicate[T <: TargetUser](

predicate: NamedPredicate[T]

)(

implicit stats: StatsReceiver

): NamedPredicate[T] = {

Predicate

.fromAsync { t: T => t.isTeamMember }

.or(predicate)

.withStats(stats.scope(predicate.name))

.withName(predicate.name)

}

def targetValidMobileSDKPredicate[T <: Target](

implicit statsReceiver: StatsReceiver

): NamedPredicate[T] = {

val name = "valid\_mobile\_sdk"

val scopedStats = statsReceiver.scope(s"targetpredicate\_$name")

Predicate

.fromAsync { target: T =>

TargetUserPredicates.validMobileSDKPredicate

.apply(Seq(target)).map(\_.headOption.getOrElse(false))

}.withStats(scopedStats)

.withName(name)

}

def magicRecsMinDurationSinceSent[T <: Target](

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[T] = {

val name = "target\_min\_duration\_since\_push"

Predicate

.fromAsync { target: T =>

PushCapUtil.getMinDurationSincePush(target, statsReceiver).flatMap { minDurationSincePush =>

CommonFatiguePredicate

.magicRecsMinDurationSincePush(interval = minDurationSincePush)

.apply(Seq(target)).map(\_.head)

}

}

.withStats(statsReceiver.scope(name))

.withName(name)

}

def optoutProbPredicate[

T <: TargetUser with TargetABDecider with TargetScoringDetails with FrigateHistory

](

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[T] = {

val name = "target\_has\_high\_optout\_probability"

Predicate

.fromAsync { target: T =>

val isNewUser = target.is30DayNewUserFromSnowflakeIdTime

if (isNewUser) {

statsReceiver.scope(name).counter("all\_new\_users").incr()

}

target.bucketOptoutProbability

.flatMap {

case Some(optoutProb) =>

if (optoutProb >= target.params(PushFeatureSwitchParams.BucketOptoutThresholdParam)) {

CommonFatiguePredicate

.magicRecsPushTargetFatiguePredicate(

interval = 24.hours,

maxInInterval = target.params(PushFeatureSwitchParams.OptoutExptPushCapParam)

)

.apply(Seq(target))

.map { values =>

val isValid = values.headOption.getOrElse(false)

if (!isValid && isNewUser) {

statsReceiver.scope(name).counter("filtered\_new\_users").incr()

}

isValid

}

} else Future.True

case \_ => Future.True

}

}

.withStats(statsReceiver.scope(name))

.withName(name)

}

/\*\*

\* Predicate used to specify CRT fatigue given interval and max number of candidates within interval.

\* @param crt The specific CRT that this predicate is being applied to

\* @param intervalParam The fatigue interval

\* @param maxInIntervalParam The max number of the given CRT's candidates that are acceptable

\* in the interval

\* @param stats StatsReceiver

\* @return Target Predicate

\*/

def pushRecTypeFatiguePredicate(

crt: CRT,

intervalParam: Param[Duration],

maxInIntervalParam: FSBoundedParam[Int],

stats: StatsReceiver

): Predicate[Target] =

Predicate.fromAsync { target: Target =>

val interval = target.params(intervalParam)

val maxIninterval = target.params(maxInIntervalParam)

CommonFatiguePredicate

.recTypeTargetFatiguePredicate(

interval = interval,

maxInInterval = maxIninterval,

recommendationType = crt,

notificationDisplayLocation = NotificationDisplayLocation.PushToMobileDevice,

minInterval = 30.minutes

)(stats.scope(s"${crt}\_push\_candidate\_fatigue")).apply(Seq(target)).map(\_.head)

}

def inlineActionFatiguePredicate(

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[Target] = {

val name = "inline\_action\_fatigue"

val predicateRequests = statsReceiver.scope(name).counter("requests")

val targetIsInExpt = statsReceiver.scope(name).counter("target\_in\_expt")

val predicateEnabled = statsReceiver.scope(name).counter("enabled")

val predicateDisabled = statsReceiver.scope(name).counter("disabled")

val inlineFatigueDisabled = statsReceiver.scope(name).counter("inline\_fatigue\_disabled")

Predicate

.fromAsync { target: Target =>

predicateRequests.incr()

if (target.params(PushFeatureSwitchParams.TargetInInlineActionAppVisitFatigue)) {

targetIsInExpt.incr()

target.inlineActionHistory.map { inlineHistory =>

if (inlineHistory.nonEmpty && target.params(

PushFeatureSwitchParams.EnableInlineActionAppVisitFatigue)) {

predicateEnabled.incr()

val inlineFatigue = target.params(PushFeatureSwitchParams.InlineActionAppVisitFatigue)

val lookbackInMs = inlineFatigue.ago.inMilliseconds

val filteredHistory = inlineHistory.filter {

case (time, \_) => time > lookbackInMs

}

filteredHistory.isEmpty

} else {

inlineFatigueDisabled.incr()

true

}

}

} else {

predicateDisabled.incr()

Future.True

}

}

.withStats(statsReceiver.scope(name))

.withName(name)

}

def webNotifsHoldback[T <: TargetUser with UserDetails with TargetABDecider](

)(

implicit stats: StatsReceiver

): NamedPredicate[T] = {

val name = "mr\_web\_notifs\_holdback"

Predicate

.fromAsync { targetUserContext: T =>

targetUserContext.deviceInfo.map { deviceInfoOpt =>

val isPrimaryWeb = deviceInfoOpt.exists {

\_.guessedPrimaryClient.exists { clientType =>

clientType == MobileClientType.Web

}

}

!(isPrimaryWeb && targetUserContext.params(PushFeatureSwitchParams.MRWebHoldbackParam))

}

}

.withStats(stats.scope(s"predicate\_$name"))

.withName(name)

}

}