package com.twitter.frigate.pushservice.predicate.event

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.EventCandidate

import com.twitter.frigate.common.base.TargetInfo

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

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

import com.twitter.frigate.common.history.RecItems

import com.twitter.frigate.magic\_events.thriftscala.Locale

import com.twitter.frigate.pushservice.model.MagicFanoutEventHydratedCandidate

import com.twitter.frigate.pushservice.model.MagicFanoutEventPushCandidate

import com.twitter.frigate.pushservice.model.MagicFanoutNewsEventPushCandidate

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

import com.twitter.frigate.pushservice.predicate.magic\_fanout.MagicFanoutPredicatesUtil.\_

import com.twitter.frigate.thriftscala.CommonRecommendationType

import com.twitter.hermit.predicate.NamedPredicate

import com.twitter.hermit.predicate.Predicate

import com.twitter.util.Future

object EventPredicatesForCandidate {

def hasTitle(

implicit statsReceiver: StatsReceiver

): NamedPredicate[MagicFanoutEventHydratedCandidate] = {

val name = "event\_title\_available"

val scopedStatsReceiver = statsReceiver.scope(s"predicate\_$name")

Predicate

.fromAsync { candidate: MagicFanoutEventHydratedCandidate =>

candidate.eventTitleFut.map(\_.nonEmpty)

}

.withStats(scopedStatsReceiver)

.withName(name)

}

def isNotDuplicateWithEventId(

implicit statsReceiver: StatsReceiver

): NamedPredicate[MagicFanoutEventHydratedCandidate] = {

val name = "duplicate\_event\_id"

Predicate

.fromAsync { candidate: MagicFanoutEventHydratedCandidate =>

val useRelaxedFatigueLengthFut: Future[Boolean] =

candidate match {

case mfNewsEvent: MagicFanoutNewsEventPushCandidate =>

mfNewsEvent.isHighPriorityEvent

case \_ => Future.value(false)

}

Future.join(candidate.target.history, useRelaxedFatigueLengthFut).map {

case (history, useRelaxedFatigueLength) =>

val filteredNotifications = if (useRelaxedFatigueLength) {

val relaxedFatigueInterval =

candidate.target

.params(

PushFeatureSwitchParams.MagicFanoutRelaxedEventIdFatigueIntervalInHours).hours

history.notificationMap.filterKeys { time =>

time.untilNow <= relaxedFatigueInterval

}.values

} else history.notificationMap.values

!RecItems(filteredNotifications.toSeq).events.exists(\_.eventId == candidate.eventId)

}

}

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

.withName(name)

}

def isNotDuplicateWithEventIdForCandidate[

T <: TargetUser with FrigateHistory,

Cand <: EventCandidate with TargetInfo[T]

](

implicit statsReceiver: StatsReceiver

): NamedPredicate[Cand] = {

val name = "is\_not\_duplicate\_event"

Predicate

.fromAsync { candidate: Cand =>

candidate.target.pushRecItems.map {

!\_.events.map(\_.eventId).contains(candidate.eventId)

}

}

.withStats(statsReceiver.scope(name))

.withName(name)

}

def accountCountryPredicateWithAllowlist(

implicit stats: StatsReceiver

): NamedPredicate[MagicFanoutEventPushCandidate] = {

val name = "account\_country\_predicate\_with\_allowlist"

val scopedStats = stats.scope(name)

val skipPredicate = Predicate

.from { candidate: MagicFanoutEventPushCandidate =>

candidate.target.params(PushFeatureSwitchParams.MagicFanoutSkipAccountCountryPredicate)

}

.withStats(stats.scope("skip\_account\_country\_predicate\_mf"))

.withName("skip\_account\_country\_predicate\_mf")

val excludeEventFromAccountCountryPredicateFiltering = Predicate

.from { candidate: MagicFanoutEventPushCandidate =>

val eventId = candidate.eventId

val target = candidate.target

target

.params(PushFeatureSwitchParams.MagicFanoutEventAllowlistToSkipAccountCountryPredicate)

.exists(eventId.equals)

}

.withStats(stats.scope("exclude\_event\_from\_account\_country\_predicate\_filtering"))

.withName("exclude\_event\_from\_account\_country\_predicate\_filtering")

skipPredicate

.or(excludeEventFromAccountCountryPredicateFiltering)

.or(accountCountryPredicate)

.withStats(scopedStats)

.withName(name)

}

/\*\*

\* Check if user's country is targeted

\* @param stats

\*/

def accountCountryPredicate(

implicit stats: StatsReceiver

): NamedPredicate[MagicFanoutEventPushCandidate] = {

val name = "account\_country\_predicate"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

val internationalLocalePassedCounter =

scopedStatsReceiver.counter("international\_locale\_passed")

val internationalLocaleFilteredCounter =

scopedStatsReceiver.counter("international\_locale\_filtered")

Predicate

.fromAsync { candidate: MagicFanoutEventPushCandidate =>

candidate.target.countryCode.map {

case Some(countryCode) =>

val denyListedCountryCodes: Seq[String] =

if (candidate.commonRecType == CommonRecommendationType.MagicFanoutNewsEvent) {

candidate.target

.params(PushFeatureSwitchParams.MagicFanoutDenyListedCountries)

} else if (candidate.commonRecType == CommonRecommendationType.MagicFanoutSportsEvent) {

candidate.target

.params(PushFeatureSwitchParams.MagicFanoutSportsEventDenyListedCountries)

} else Seq()

val eventCountries =

candidate.newsForYouMetadata

.flatMap(\_.locales).getOrElse(Seq.empty[Locale]).flatMap(\_.country)

if (isInCountryList(countryCode, eventCountries)

&& !isInCountryList(countryCode, denyListedCountryCodes)) {

internationalLocalePassedCounter.incr()

true

} else {

internationalLocaleFilteredCounter.incr()

false

}

case \_ => false

}

}

.withStats(scopedStatsReceiver)

.withName(name)

}

}