package com.twitter.frigate.pushservice.predicate

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.SocialGraphServiceRelationshipMap

import com.twitter.frigate.common.base.TweetAuthor

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

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

import com.twitter.frigate.pushservice.params.PushParams

import com.twitter.gizmoduck.thriftscala.UserType

import com.twitter.hermit.predicate.NamedPredicate

import com.twitter.hermit.predicate.Predicate

import com.twitter.hermit.predicate.socialgraph.Edge

import com.twitter.hermit.predicate.socialgraph.RelationEdge

import com.twitter.socialgraph.thriftscala.RelationshipType

import com.twitter.util.Future

/\*\*

\* Refactor SGS predicates so that predicates can use relationshipMap we generate in hydrate step

\*/

object SGSPredicatesForCandidate {

case class RelationshipMapEdge(edge: Edge, relationshipMap: Map[RelationEdge, Boolean])

private def relationshipMapEdgeFromCandidate(

candidate: PushCandidate with TweetAuthor with SocialGraphServiceRelationshipMap

): Option[RelationshipMapEdge] = {

candidate.authorId map { authorId =>

RelationshipMapEdge(Edge(candidate.target.targetId, authorId), candidate.relationshipMap)

}

}

def authorBeingFollowed(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with TweetAuthor with SocialGraphServiceRelationshipMap] = {

val name = "author\_not\_being\_followed"

val stats = statsReceiver.scope(name)

val softUserCounter = stats.counter("soft\_user")

val sgsAuthorBeingFollowedPredicate = Predicate

.from { relationshipMapEdge: RelationshipMapEdge =>

anyRelationExist(relationshipMapEdge, Set(RelationshipType.Following))

}

Predicate

.fromAsync {

candidate: PushCandidate with TweetAuthor with SocialGraphServiceRelationshipMap =>

val target = candidate.target

target.targetUser.flatMap {

case Some(gizmoduckUser) if gizmoduckUser.userType == UserType.Soft =>

softUserCounter.incr()

target.seedsWithWeight.map { followedUsersWithWeightOpt =>

candidate.authorId match {

case Some(authorId) =>

val followedUsers = followedUsersWithWeightOpt.getOrElse(Map.empty).keys

followedUsers.toSet.contains(authorId)

case None => false

}

}

case \_ =>

sgsAuthorBeingFollowedPredicate

.optionalOn(relationshipMapEdgeFromCandidate, missingResult = false)

.apply(Seq(candidate))

.map(\_.head)

}

}.withStats(stats)

.withName(name)

}

def authorNotBeingDeviceFollowed(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with TweetAuthor with SocialGraphServiceRelationshipMap] = {

val name = "author\_being\_device\_followed"

Predicate

.from { relationshipMapEdge: RelationshipMapEdge =>

{

anyRelationExist(relationshipMapEdge, Set(RelationshipType.DeviceFollowing))

}

}

.optionalOn(relationshipMapEdgeFromCandidate, missingResult = false)

.flip

.withStats(statsReceiver.scope(name))

.withName(name)

}

def recommendedTweetAuthorAcceptableToTargetUser(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with TweetAuthor with SocialGraphServiceRelationshipMap] = {

val name = "recommended\_tweet\_author\_not\_acceptable\_to\_target\_user"

Predicate

.from { relationshipMapEdge: RelationshipMapEdge =>

{

anyRelationExist(

relationshipMapEdge,

Set(

RelationshipType.Blocking,

RelationshipType.BlockedBy,

RelationshipType.HideRecommendations,

RelationshipType.Muting

))

}

}

.flip

.optionalOn(relationshipMapEdgeFromCandidate, missingResult = false)

.withStats(statsReceiver.scope(name))

.withName(name)

}

def authorNotBeingFollowed(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with TweetAuthor with SocialGraphServiceRelationshipMap] = {

Predicate

.from { relationshipMapEdge: RelationshipMapEdge =>

{

anyRelationExist(relationshipMapEdge, Set(RelationshipType.Following))

}

}

.optionalOn(relationshipMapEdgeFromCandidate, missingResult = false)

.flip

.withStats(statsReceiver.scope("predicate\_author\_not\_being\_followed\_pre\_ranking"))

.withName("author\_not\_being\_followed")

}

def disableInNetworkTweetPredicate(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with TweetAuthor with SocialGraphServiceRelationshipMap] = {

val name = "enable\_in\_network\_tweet"

Predicate

.fromAsync {

candidate: PushCandidate with TweetAuthor with SocialGraphServiceRelationshipMap =>

if (candidate.target.params(PushParams.DisableInNetworkTweetCandidatesParam)) {

authorNotBeingFollowed

.apply(Seq(candidate))

.map(\_.head)

} else Future.True

}.withStats(statsReceiver.scope(name))

.withName(name)

}

def disableOutNetworkTweetPredicate(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with TweetAuthor with SocialGraphServiceRelationshipMap] = {

val name = "enable\_out\_network\_tweet"

Predicate

.fromAsync {

candidate: PushCandidate with TweetAuthor with SocialGraphServiceRelationshipMap =>

if (candidate.target.params(PushFeatureSwitchParams.DisableOutNetworkTweetCandidatesFS)) {

authorBeingFollowed

.apply(Seq(candidate))

.map(\_.head)

} else Future.True

}.withStats(statsReceiver.scope(name))

.withName(name)

}

/\*\*

\* Returns true if the provided relationshipEdge exists among

\* @param candidate candidate

\* @param relationships relaionships

\* @return Boolean result

\*/

private def anyRelationExist(

relationshipMapEdge: RelationshipMapEdge,

relationships: Set[RelationshipType]

): Boolean = {

val resultSeq = relationships.map { relationship =>

relationshipMapEdge.relationshipMap.getOrElse(

RelationEdge(relationshipMapEdge.edge, relationship),

false)

}.toSeq

resultSeq.contains(true)

}

}