package com.twitter.frigate.pushservice.predicate

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.SpaceCandidate

import com.twitter.frigate.common.base.SpaceCandidateDetails

import com.twitter.frigate.common.base.TweetAuthorDetails

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

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

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

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.hermit.predicate.socialgraph.SocialGraphPredicate

import com.twitter.socialgraph.thriftscala.RelationshipType

import com.twitter.storehaus.ReadableStore

import com.twitter.strato.response.Err

import com.twitter.ubs.thriftscala.AudioSpace

import com.twitter.ubs.thriftscala.BroadcastState

import com.twitter.ubs.thriftscala.ParticipantUser

import com.twitter.ubs.thriftscala.Participants

import com.twitter.util.Future

object SpacePredicate {

/\*\* Filters the request if the target is present in the space as a listener, speakeTestConfigr, or admin \*/

def targetInSpace(

audioSpaceParticipantsStore: ReadableStore[String, Participants]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[SpaceCandidateDetails with RawCandidate] = {

val name = "target\_in\_space"

Predicate

.fromAsync[SpaceCandidateDetails with RawCandidate] { spaceCandidate =>

audioSpaceParticipantsStore.get(spaceCandidate.spaceId).map {

case Some(participants) =>

val allParticipants: Seq[ParticipantUser] =

(participants.admins ++ participants.speakers ++ participants.listeners).flatten.toSeq

val isInSpace = allParticipants.exists { participant =>

participant.twitterUserId.contains(spaceCandidate.target.targetId)

}

!isInSpace

case None => false

}

}.withStats(statsReceiver.scope(name))

.withName(name)

}

/\*\*

\*

\* @param audioSpaceStore: space metadata store

\* @param statsReceiver: record stats

\* @return: true if the space not started ELSE false to filter out notification

\*/

def scheduledSpaceStarted(

audioSpaceStore: ReadableStore[String, AudioSpace]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[SpaceCandidate with RawCandidate] = {

val name = "scheduled\_space\_started"

Predicate

.fromAsync[SpaceCandidate with RawCandidate] { spaceCandidate =>

audioSpaceStore

.get(spaceCandidate.spaceId)

.map(\_.exists(\_.state.contains(BroadcastState.NotStarted)))

.rescue {

case Err(Err.Authorization, \_, \_) =>

Future.False

}

}

.withStats(statsReceiver.scope(name))

.withName(name)

}

private def relationshipMapEdgeFromSpaceCandidate(

candidate: RawCandidate with SpaceCandidate

): Option[(Long, Seq[Long])] = {

candidate.hostId.map { spaceHostId =>

(candidate.target.targetId, Seq(spaceHostId))

}

}

/\*\*

\* Check only host block for scheduled space reminders

\* @return: True if no blocking relation between host and target user, else False

\*/

def spaceHostTargetUserBlocking(

edgeStore: ReadableStore[RelationEdge, Boolean]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[SpaceCandidate with RawCandidate] = {

val name = "space\_host\_target\_user\_blocking"

PredicatesForCandidate

.blocking(edgeStore)

.optionalOn(relationshipMapEdgeFromSpaceCandidate, false)

.withStats(statsReceiver.scope(name))

.withName(name)

}

private def edgeFromCandidate(

candidate: PushCandidate with TweetAuthorDetails

): Future[Option[Edge]] = {

candidate.tweetAuthor.map(\_.map { author => Edge(candidate.target.targetId, author.id) })

}

def recommendedTweetAuthorAcceptableToTargetUser(

edgeStore: ReadableStore[RelationEdge, Boolean]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with TweetAuthorDetails] = {

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

SocialGraphPredicate

.anyRelationExists(

edgeStore,

Set(

RelationshipType.Blocking,

RelationshipType.BlockedBy,

RelationshipType.HideRecommendations,

RelationshipType.Muting

)

)

.flip

.flatOptionContraMap(

edgeFromCandidate,

missingResult = false

)

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

.withName(name)

}

def narrowCastSpace(

implicit statsReceiver: StatsReceiver

): NamedPredicate[SpaceCandidateDetails with RawCandidate] = {

val name = "narrow\_cast\_space"

val narrowCastSpaceScope = statsReceiver.scope(name)

val employeeSpaceCounter = narrowCastSpaceScope.counter("employees")

val superFollowerSpaceCounter = narrowCastSpaceScope.counter("super\_followers")

Predicate

.fromAsync[SpaceCandidateDetails with RawCandidate] { candidate =>

candidate.audioSpaceFut.map {

case Some(audioSpace) if audioSpace.narrowCastSpaceType.contains(1L) =>

employeeSpaceCounter.incr()

candidate.target.params(PushFeatureSwitchParams.EnableEmployeeOnlySpaceNotifications)

case Some(audioSpace) if audioSpace.narrowCastSpaceType.contains(2L) =>

superFollowerSpaceCounter.incr()

false

case \_ => true

}

}.withStats(narrowCastSpaceScope)

.withName(name)

}

}