package com.twitter.frigate.pushservice.util

import com.twitter.channels.common.thriftscala.ApiList

import com.twitter.escherbird.common.thriftscala.Domains

import com.twitter.escherbird.metadata.thriftscala.EntityMegadata

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.\_

import com.twitter.frigate.common.store.interests.InterestsLookupRequestWithContext

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

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

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

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

import com.twitter.frigate.pushservice.model.\_

import com.twitter.frigate.pushservice.model.FanoutReasonEntities

import com.twitter.frigate.pushservice.ml.PushMLModelScorer

import com.twitter.frigate.pushservice.model.candidate.CopyIds

import com.twitter.frigate.pushservice.store.EventRequest

import com.twitter.frigate.pushservice.store.UttEntityHydrationStore

import com.twitter.gizmoduck.thriftscala.User

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

import com.twitter.hermit.store.semantic\_core.SemanticEntityForQuery

import com.twitter.interests.thriftscala.UserInterests

import com.twitter.livevideo.timeline.domain.v2.{Event => LiveEvent}

import com.twitter.simclusters\_v2.thriftscala.SimClustersInferredEntities

import com.twitter.storehaus.FutureOps

import com.twitter.storehaus.ReadableStore

import com.twitter.strato.client.UserId

import com.twitter.ubs.thriftscala.AudioSpace

import com.twitter.util.Future

object CandidateHydrationUtil {

def getAuthorIdFromTweetCandidate(tweetCandidate: TweetCandidate): Option[Long] = {

tweetCandidate match {

case candidate: TweetCandidate with TweetAuthor =>

candidate.authorId

case \_ => None

}

}

private def getCandidateAuthorFromUserMap(

tweetCandidate: TweetCandidate,

userMap: Map[Long, User]

): Option[User] = {

getAuthorIdFromTweetCandidate(tweetCandidate) match {

case Some(id) =>

userMap.get(id)

case \_ =>

None

}

}

private def getRelationshipMapForInNetworkCandidate(

candidate: RawCandidate with TweetAuthor,

relationshipMap: Map[RelationEdge, Boolean]

): Map[RelationEdge, Boolean] = {

val relationEdges =

RelationshipUtil.getPreCandidateRelationshipsForInNetworkTweets(candidate).toSet

relationEdges.map { relationEdge =>

(relationEdge, relationshipMap(relationEdge))

}.toMap

}

private def getTweetCandidateSocialContextUsers(

candidate: RawCandidate with SocialContextActions,

userMap: Map[Long, User]

): Map[Long, Option[User]] = {

candidate.socialContextUserIds.map { userId => userId -> userMap.get(userId) }.toMap

}

type TweetWithSocialContextTraits = TweetCandidate with TweetDetails with SocialContextActions

def getHydratedCandidateForTweetRetweet(

candidate: RawCandidate with TweetWithSocialContextTraits,

userMap: Map[Long, User],

copyIds: CopyIds

)(

implicit stats: StatsReceiver,

pushModelScorer: PushMLModelScorer

): TweetRetweetPushCandidate = {

new TweetRetweetPushCandidate(

candidate = candidate,

socialContextUserMap = Future.value(getTweetCandidateSocialContextUsers(candidate, userMap)),

author = Future.value(getCandidateAuthorFromUserMap(candidate, userMap)),

copyIds: CopyIds

)

}

def getHydratedCandidateForTweetFavorite(

candidate: RawCandidate with TweetWithSocialContextTraits,

userMap: Map[Long, User],

copyIds: CopyIds

)(

implicit stats: StatsReceiver,

pushModelScorer: PushMLModelScorer

): TweetFavoritePushCandidate = {

new TweetFavoritePushCandidate(

candidate = candidate,

socialContextUserMap = Future.value(getTweetCandidateSocialContextUsers(candidate, userMap)),

author = Future.value(getCandidateAuthorFromUserMap(candidate, userMap)),

copyIds = copyIds

)

}

def getHydratedCandidateForF1FirstDegreeTweet(

candidate: RawCandidate with F1FirstDegree,

userMap: Map[Long, User],

relationshipMap: Map[RelationEdge, Boolean],

copyIds: CopyIds

)(

implicit stats: StatsReceiver,

pushModelScorer: PushMLModelScorer

): F1TweetPushCandidate = {

new F1TweetPushCandidate(

candidate = candidate,

author = Future.value(getCandidateAuthorFromUserMap(candidate, userMap)),

socialGraphServiceResultMap =

getRelationshipMapForInNetworkCandidate(candidate, relationshipMap),

copyIds = copyIds

)

}

def getHydratedTopicProofTweetCandidate(

candidate: RawCandidate with TopicProofTweetCandidate,

userMap: Map[Long, User],

copyIds: CopyIds

)(

implicit stats: StatsReceiver,

pushMLModelScorer: PushMLModelScorer

): TopicProofTweetPushCandidate =

new TopicProofTweetPushCandidate(

candidate,

getCandidateAuthorFromUserMap(candidate, userMap),

copyIds

)

def getHydratedSubscribedSearchTweetCandidate(

candidate: RawCandidate with SubscribedSearchTweetCandidate,

userMap: Map[Long, User],

copyIds: CopyIds

)(

implicit stats: StatsReceiver,

pushMLModelScorer: PushMLModelScorer

): SubscribedSearchTweetPushCandidate =

new SubscribedSearchTweetPushCandidate(

candidate,

getCandidateAuthorFromUserMap(candidate, userMap),

copyIds)

def getHydratedListCandidate(

apiListStore: ReadableStore[Long, ApiList],

candidate: RawCandidate with ListPushCandidate,

copyIds: CopyIds

)(

implicit stats: StatsReceiver,

pushMLModelScorer: PushMLModelScorer

): ListRecommendationPushCandidate = {

new ListRecommendationPushCandidate(apiListStore, candidate, copyIds)

}

def getHydratedCandidateForOutOfNetworkTweetCandidate(

candidate: RawCandidate with OutOfNetworkTweetCandidate with TopicCandidate,

userMap: Map[Long, User],

copyIds: CopyIds

)(

implicit stats: StatsReceiver,

pushModelScorer: PushMLModelScorer

): OutOfNetworkTweetPushCandidate = {

new OutOfNetworkTweetPushCandidate(

candidate: RawCandidate with OutOfNetworkTweetCandidate with TopicCandidate,

author = Future.value(getCandidateAuthorFromUserMap(candidate, userMap)),

copyIds: CopyIds

)

}

def getHydratedCandidateForTripTweetCandidate(

candidate: RawCandidate with OutOfNetworkTweetCandidate with TripCandidate,

userMap: Map[Long, User],

copyIds: CopyIds

)(

implicit stats: StatsReceiver,

pushModelScorer: PushMLModelScorer

): TripTweetPushCandidate = {

new TripTweetPushCandidate(

candidate: RawCandidate with OutOfNetworkTweetCandidate with TripCandidate,

author = Future.value(getCandidateAuthorFromUserMap(candidate, userMap)),

copyIds: CopyIds

)

}

def getHydratedCandidateForDiscoverTwitterCandidate(

candidate: RawCandidate with DiscoverTwitterCandidate,

copyIds: CopyIds

)(

implicit stats: StatsReceiver,

pushModelScorer: PushMLModelScorer

): DiscoverTwitterPushCandidate = {

new DiscoverTwitterPushCandidate(

candidate = candidate,

copyIds = copyIds

)

}

/\*\*

\* /\*

\* This method can be reusable for hydrating event candidates

\*\*/

\* @param candidate

\* @param fanoutMetadataStore

\* @param semanticCoreMegadataStore

\* @return (hydratedEvent, hydratedFanoutEvent, hydratedSemanticEntityResults, hydratedSemanticCoreMegadata)

\*/

private def hydrateMagicFanoutEventCandidate(

candidate: RawCandidate with MagicFanoutEventCandidate,

fanoutMetadataStore: ReadableStore[(Long, Long), FanoutEvent],

semanticCoreMegadataStore: ReadableStore[SemanticEntityForQuery, EntityMegadata]

): Future[MagicFanoutEventHydratedInfo] = {

val fanoutEventFut = fanoutMetadataStore.get((candidate.eventId, candidate.pushId))

val semanticEntityForQueries: Seq[SemanticEntityForQuery] = {

val semanticCoreEntityIdQueries = candidate.candidateMagicEventsReasons match {

case magicEventsReasons: Seq[MagicEventsReason] =>

magicEventsReasons.map(\_.reason).collect {

case TargetID.SemanticCoreID(scInterest) =>

SemanticEntityForQuery(domainId = scInterest.domainId, entityId = scInterest.entityId)

}

case \_ => Seq.empty

}

val eventEntityQuery = SemanticEntityForQuery(

domainId = Domains.EventsEntityService.value,

entityId = candidate.eventId)

semanticCoreEntityIdQueries :+ eventEntityQuery

}

val semanticEntityResultsFut = FutureOps.mapCollect(

semanticCoreMegadataStore.multiGet(semanticEntityForQueries.toSet)

)

Future

.join(fanoutEventFut, semanticEntityResultsFut).map {

case (fanoutEvent, semanticEntityResults) =>

MagicFanoutEventHydratedInfo(

fanoutEvent,

semanticEntityResults

)

case \_ =>

throw new IllegalArgumentException(

"event candidate hydration errors" + candidate.frigateNotification.toString)

}

}

def getHydratedCandidateForMagicFanoutNewsEvent(

candidate: RawCandidate with MagicFanoutNewsEventCandidate,

copyIds: CopyIds,

lexServiceStore: ReadableStore[EventRequest, LiveEvent],

fanoutMetadataStore: ReadableStore[(Long, Long), FanoutEvent],

semanticCoreMegadataStore: ReadableStore[SemanticEntityForQuery, EntityMegadata],

simClusterToEntityStore: ReadableStore[Int, SimClustersInferredEntities],

interestsLookupStore: ReadableStore[InterestsLookupRequestWithContext, UserInterests],

uttEntityHydrationStore: UttEntityHydrationStore

)(

implicit stats: StatsReceiver,

pushModelScorer: PushMLModelScorer

): Future[MagicFanoutNewsEventPushCandidate] = {

val magicFanoutEventHydratedInfoFut = hydrateMagicFanoutEventCandidate(

candidate,

fanoutMetadataStore,

semanticCoreMegadataStore

)

lazy val simClusterToEntityMappingFut: Future[Map[Int, Option[SimClustersInferredEntities]]] =

Future.collect {

simClusterToEntityStore.multiGet(

FanoutReasonEntities

.from(candidate.candidateMagicEventsReasons.map(\_.reason)).simclusterIds.map(

\_.clusterId)

)

}

Future

.join(

magicFanoutEventHydratedInfoFut,

simClusterToEntityMappingFut

).map {

case (magicFanoutEventHydratedInfo, simClusterToEntityMapping) =>

new MagicFanoutNewsEventPushCandidate(

candidate = candidate,

copyIds = copyIds,

fanoutEvent = magicFanoutEventHydratedInfo.fanoutEvent,

semanticEntityResults = magicFanoutEventHydratedInfo.semanticEntityResults,

simClusterToEntities = simClusterToEntityMapping,

lexServiceStore = lexServiceStore,

interestsLookupStore = interestsLookupStore,

uttEntityHydrationStore = uttEntityHydrationStore

)

}

}

def getHydratedCandidateForMagicFanoutSportsEvent(

candidate: RawCandidate

with MagicFanoutSportsEventCandidate

with MagicFanoutSportsScoreInformation,

copyIds: CopyIds,

lexServiceStore: ReadableStore[EventRequest, LiveEvent],

fanoutMetadataStore: ReadableStore[(Long, Long), FanoutEvent],

semanticCoreMegadataStore: ReadableStore[SemanticEntityForQuery, EntityMegadata],

interestsLookupStore: ReadableStore[InterestsLookupRequestWithContext, UserInterests],

uttEntityHydrationStore: UttEntityHydrationStore

)(

implicit stats: StatsReceiver,

pushModelScorer: PushMLModelScorer

): Future[MagicFanoutSportsPushCandidate] = {

val magicFanoutEventHydratedInfoFut = hydrateMagicFanoutEventCandidate(

candidate,

fanoutMetadataStore,

semanticCoreMegadataStore

)

magicFanoutEventHydratedInfoFut.map { magicFanoutEventHydratedInfo =>

new MagicFanoutSportsPushCandidate(

candidate = candidate,

copyIds = copyIds,

fanoutEvent = magicFanoutEventHydratedInfo.fanoutEvent,

semanticEntityResults = magicFanoutEventHydratedInfo.semanticEntityResults,

simClusterToEntities = Map.empty,

lexServiceStore = lexServiceStore,

interestsLookupStore = interestsLookupStore,

uttEntityHydrationStore = uttEntityHydrationStore

)

}

}

def getHydratedCandidateForMagicFanoutProductLaunch(

candidate: RawCandidate with MagicFanoutProductLaunchCandidate,

copyIds: CopyIds

)(

implicit stats: StatsReceiver,

pushModelScorer: PushMLModelScorer

): Future[MagicFanoutProductLaunchPushCandidate] =

Future.value(new MagicFanoutProductLaunchPushCandidate(candidate, copyIds))

def getHydratedCandidateForMagicFanoutCreatorEvent(

candidate: RawCandidate with MagicFanoutCreatorEventCandidate,

safeUserStore: ReadableStore[Long, User],

copyIds: CopyIds,

creatorTweetCountStore: ReadableStore[UserId, Int]

)(

implicit stats: StatsReceiver,

pushModelScorer: PushMLModelScorer

): Future[MagicFanoutCreatorEventPushCandidate] = {

safeUserStore.get(candidate.creatorId).map { hydratedCreatorUser =>

new MagicFanoutCreatorEventPushCandidate(

candidate,

hydratedCreatorUser,

copyIds,

creatorTweetCountStore)

}

}

def getHydratedCandidateForScheduledSpaceSubscriber(

candidate: RawCandidate with ScheduledSpaceSubscriberCandidate,

safeUserStore: ReadableStore[Long, User],

copyIds: CopyIds,

audioSpaceStore: ReadableStore[String, AudioSpace]

)(

implicit stats: StatsReceiver,

pushModelScorer: PushMLModelScorer

): Future[ScheduledSpaceSubscriberPushCandidate] = {

candidate.hostId match {

case Some(spaceHostId) =>

safeUserStore.get(spaceHostId).map { hydratedHost =>

new ScheduledSpaceSubscriberPushCandidate(

candidate = candidate,

hostUser = hydratedHost,

copyIds = copyIds,

audioSpaceStore = audioSpaceStore

)

}

case \_ =>

Future.exception(

new IllegalStateException(

"Missing Space Host Id for hydrating ScheduledSpaceSubscriberCandidate"))

}

}

def getHydratedCandidateForScheduledSpaceSpeaker(

candidate: RawCandidate with ScheduledSpaceSpeakerCandidate,

safeUserStore: ReadableStore[Long, User],

copyIds: CopyIds,

audioSpaceStore: ReadableStore[String, AudioSpace]

)(

implicit stats: StatsReceiver,

pushModelScorer: PushMLModelScorer

): Future[ScheduledSpaceSpeakerPushCandidate] = {

candidate.hostId match {

case Some(spaceHostId) =>

safeUserStore.get(spaceHostId).map { hydratedHost =>

new ScheduledSpaceSpeakerPushCandidate(

candidate = candidate,

hostUser = hydratedHost,

copyIds = copyIds,

audioSpaceStore = audioSpaceStore

)

}

case \_ =>

Future.exception(

new RuntimeException(

"Missing Space Host Id for hydrating ScheduledSpaceSpeakerCandidate"))

}

}

def getHydratedCandidateForTopTweetImpressionsCandidate(

candidate: RawCandidate with TopTweetImpressionsCandidate,

copyIds: CopyIds

)(

implicit stats: StatsReceiver,

pushModelScorer: PushMLModelScorer

): TopTweetImpressionsPushCandidate = {

new TopTweetImpressionsPushCandidate(

candidate = candidate,

copyIds = copyIds

)

}

def isNsfwAccount(user: User, nsfwTokens: Seq[String]): Boolean = {

def hasNsfwToken(str: String): Boolean = nsfwTokens.exists(str.toLowerCase().contains(\_))

val name = user.profile.map(\_.name).getOrElse("")

val screenName = user.profile.map(\_.screenName).getOrElse("")

val location = user.profile.map(\_.location).getOrElse("")

val description = user.profile.map(\_.description).getOrElse("")

val hasNsfwFlag =

user.safety.map(safety => safety.nsfwUser || safety.nsfwAdmin).getOrElse(false)

hasNsfwToken(name) || hasNsfwToken(screenName) || hasNsfwToken(location) || hasNsfwToken(

description) || hasNsfwFlag

}

}