package com.twitter.frigate.pushservice.model

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

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.MagicFanoutEventCandidate

import com.twitter.frigate.common.base.RecommendationType

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

import com.twitter.frigate.common.util.HighPriorityLocaleUtil

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

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

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

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

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

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

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

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

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

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

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

import com.twitter.frigate.pushservice.model.ibis.Ibis2HydratorForCandidate

import com.twitter.frigate.pushservice.model.ntab.EventNTabRequestHydrator

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

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

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

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

import com.twitter.frigate.pushservice.util.PushDeviceUtil

import com.twitter.frigate.pushservice.util.TopicsUtil

import com.twitter.frigate.thriftscala.FrigateNotification

import com.twitter.frigate.thriftscala.MagicFanoutEventNotificationDetails

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

import com.twitter.interests.thriftscala.InterestId.SemanticCore

import com.twitter.interests.thriftscala.UserInterests

import com.twitter.livevideo.common.ids.CountryId

import com.twitter.livevideo.common.ids.UserId

import com.twitter.livevideo.timeline.domain.v2.Event

import com.twitter.livevideo.timeline.domain.v2.HydrationOptions

import com.twitter.livevideo.timeline.domain.v2.LookupContext

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

import com.twitter.storehaus.ReadableStore

import com.twitter.topiclisting.utt.LocalizedEntity

import com.twitter.util.Future

abstract class MagicFanoutEventPushCandidate(

candidate: RawCandidate with MagicFanoutEventCandidate with RecommendationType,

copyIds: CopyIds,

override val fanoutEvent: Option[FanoutEvent],

override val semanticEntityResults: Map[SemanticEntityForQuery, Option[EntityMegadata]],

simClusterToEntities: Map[Int, Option[SimClustersInferredEntities]],

lexServiceStore: ReadableStore[EventRequest, Event],

interestsLookupStore: ReadableStore[InterestsLookupRequestWithContext, UserInterests],

uttEntityHydrationStore: UttEntityHydrationStore

)(

implicit statsScoped: StatsReceiver,

pushModelScorer: PushMLModelScorer)

extends PushCandidate

with MagicFanoutEventHydratedCandidate

with MagicFanoutEventCandidate

with EventNTabRequestHydrator

with RecommendationType

with Ibis2HydratorForCandidate {

override lazy val eventFut: Future[Option[Event]] = {

eventRequestFut.flatMap {

case Some(eventRequest) => lexServiceStore.get(eventRequest)

case \_ => Future.None

}

}

override val frigateNotification: FrigateNotification = candidate.frigateNotification

override val pushId: Long = candidate.pushId

override val candidateMagicEventsReasons: Seq[MagicEventsReason] =

candidate.candidateMagicEventsReasons

override val eventId: Long = candidate.eventId

override val momentId: Option[Long] = candidate.momentId

override val target: Target = candidate.target

override val eventLanguage: Option[String] = candidate.eventLanguage

override val details: Option[MagicFanoutEventNotificationDetails] = candidate.details

override lazy val stats: StatsReceiver = statsScoped.scope("MagicFanoutEventPushCandidate")

override val weightedOpenOrNtabClickModelScorer: PushMLModelScorer = pushModelScorer

override val pushCopyId: Option[Int] = copyIds.pushCopyId

override val ntabCopyId: Option[Int] = copyIds.ntabCopyId

override val copyAggregationId: Option[String] = copyIds.aggregationId

override val statsReceiver: StatsReceiver = statsScoped.scope("MagicFanoutEventPushCandidate")

override val effectiveMagicEventsReasons: Option[Seq[MagicEventsReason]] = Some(

candidateMagicEventsReasons)

lazy val newsForYouMetadata: Option[NewsForYouMetadata] =

fanoutEvent.flatMap { event =>

{

event.fanoutMetadata.collect {

case FanoutMetadata.NewsForYouMetadata(nfyMetadata) => nfyMetadata

}

}

}

val reverseIndexedTopicIds = candidate.candidateMagicEventsReasons

.filter(\_.source.contains(ReasonSource.UttTopicFollowGraph))

.map(\_.reason).collect {

case TargetID.SemanticCoreID(semanticCoreID) => semanticCoreID.entityId

}.toSet

val ergSemanticCoreIds = candidate.candidateMagicEventsReasons

.filter(\_.source.contains(ReasonSource.ErgShortTermInterestSemanticCore)).map(

\_.reason).collect {

case TargetID.SemanticCoreID(semanticCoreID) => semanticCoreID.entityId

}.toSet

override lazy val ergLocalizedEntities = TopicsUtil

.getLocalizedEntityMap(target, ergSemanticCoreIds, uttEntityHydrationStore)

.map { localizedEntityMap =>

ergSemanticCoreIds.collect {

case topicId if localizedEntityMap.contains(topicId) => localizedEntityMap(topicId)

}

}

val eventSemanticCoreEntityIds: Seq[Long] = {

val entityIds = for {

event <- fanoutEvent

targets <- event.targets

} yield {

targets.flatMap {

\_.whitelist.map {

\_.collect {

case TargetID.SemanticCoreID(semanticCoreID) => semanticCoreID.entityId

}

}

}

}

entityIds.map(\_.flatten).getOrElse(Seq.empty)

}

val eventSemanticCoreDomainIds: Seq[Long] = {

val domainIds = for {

event <- fanoutEvent

targets <- event.targets

} yield {

targets.flatMap {

\_.whitelist.map {

\_.collect {

case TargetID.SemanticCoreID(semanticCoreID) => semanticCoreID.domainId

}

}

}

}

domainIds.map(\_.flatten).getOrElse(Seq.empty)

}

override lazy val followedTopicLocalizedEntities: Future[Set[LocalizedEntity]] = {

val isNewSignupTargetingReason = candidateMagicEventsReasons.size == 1 &&

candidateMagicEventsReasons.headOption.exists(\_.source.contains(ReasonSource.NewSignup))

val shouldFetchTopicFollows = reverseIndexedTopicIds.nonEmpty || isNewSignupTargetingReason

val topicFollows = if (shouldFetchTopicFollows) {

TopicsUtil

.getTopicsFollowedByUser(

candidate.target,

interestsLookupStore,

stats.stat("followed\_topics")

).map { \_.getOrElse(Seq.empty) }.map {

\_.flatMap {

\_.interestId match {

case SemanticCore(semanticCore) => Some(semanticCore.id)

case \_ => None

}

}

}

} else Future.Nil

topicFollows.flatMap { followedTopicIds =>

val topicIds = if (isNewSignupTargetingReason) {

// if new signup is the only targeting reason then we check the event targeting reason

// against realtime topic follows.

eventSemanticCoreEntityIds.toSet.intersect(followedTopicIds.toSet)

} else {

// check against the fanout reason of topics

followedTopicIds.toSet.intersect(reverseIndexedTopicIds)

}

TopicsUtil

.getLocalizedEntityMap(target, topicIds, uttEntityHydrationStore)

.map { localizedEntityMap =>

topicIds.collect {

case topicId if localizedEntityMap.contains(topicId) => localizedEntityMap(topicId)

}

}

}

}

lazy val simClusterToEntityMapping: Map[Int, Seq[Long]] =

simClusterToEntities.flatMap {

case (clusterId, Some(inferredEntities)) =>

statsReceiver.counter("with\_cluster\_to\_entity\_mapping").incr()

Some(

(

clusterId,

inferredEntities.entities

.map(\_.entityId)))

case \_ =>

statsReceiver.counter("without\_cluster\_to\_entity\_mapping").incr()

None

}

lazy val annotatedAndInferredSemanticCoreEntities: Seq[Long] =

(simClusterToEntityMapping, eventFanoutReasonEntities) match {

case (entityMapping, eventFanoutReasons) =>

entityMapping.values.flatten.toSeq ++

eventFanoutReasons.semanticCoreIds.map(\_.entityId)

}

lazy val shouldHydrateSquareImage = target.deviceInfo.map { deviceInfo =>

(PushDeviceUtil.isPrimaryDeviceIOS(deviceInfo) &&

target.params(PushFeatureSwitchParams.EnableEventSquareMediaIosMagicFanoutNewsEvent)) ||

(PushDeviceUtil.isPrimaryDeviceAndroid(deviceInfo) &&

target.params(PushFeatureSwitchParams.EnableEventSquareMediaAndroid))

}

lazy val shouldHydratePrimaryImage: Future[Boolean] = target.deviceInfo.map { deviceInfo =>

(PushDeviceUtil.isPrimaryDeviceAndroid(deviceInfo) &&

target.params(PushFeatureSwitchParams.EnableEventPrimaryMediaAndroid))

}

lazy val eventRequestFut: Future[Option[EventRequest]] =

Future

.join(

target.inferredUserDeviceLanguage,

target.accountCountryCode,

shouldHydrateSquareImage,

shouldHydratePrimaryImage).map {

case (

inferredUserDeviceLanguage,

accountCountryCode,

shouldHydrateSquareImage,

shouldHydratePrimaryImage) =>

if (shouldHydrateSquareImage || shouldHydratePrimaryImage) {

Some(

EventRequest(

eventId,

lookupContext = LookupContext(

hydrationOptions = HydrationOptions(

includeSquareImage = shouldHydrateSquareImage,

includePrimaryImage = shouldHydratePrimaryImage

),

language = inferredUserDeviceLanguage,

countryCode = accountCountryCode,

userId = Some(UserId(target.targetId))

)

))

} else {

Some(

EventRequest(

eventId,

lookupContext = LookupContext(

language = inferredUserDeviceLanguage,

countryCode = accountCountryCode

)

))

}

case \_ => None

}

lazy val isHighPriorityEvent: Future[Boolean] = target.accountCountryCode.map { countryCodeOpt =>

val isHighPriorityPushOpt = for {

countryCode <- countryCodeOpt

nfyMetadata <- newsForYouMetadata

eventContext <- nfyMetadata.eventContextScribe

} yield {

val highPriorityLocales = HighPriorityLocaleUtil.getHighPriorityLocales(

eventContext = eventContext,

defaultLocalesOpt = nfyMetadata.locales)

val highPriorityGeos = HighPriorityLocaleUtil.getHighPriorityGeos(

eventContext = eventContext,

defaultGeoPlaceIdsOpt = nfyMetadata.placeIds)

val isHighPriorityLocalePush =

highPriorityLocales.flatMap(\_.country).map(CountryId(\_)).contains(CountryId(countryCode))

val isHighPriorityGeoPush = MagicFanoutPredicatesUtil

.geoPlaceIdsFromReasons(candidateMagicEventsReasons)

.intersect(highPriorityGeos.toSet)

.nonEmpty

stats.scope("is\_high\_priority\_locale\_push").counter(s"$isHighPriorityLocalePush").incr()

stats.scope("is\_high\_priority\_geo\_push").counter(s"$isHighPriorityGeoPush").incr()

isHighPriorityLocalePush || isHighPriorityGeoPush

}

isHighPriorityPushOpt.getOrElse(false)

}

}