package com.twitter.frigate.pushservice.model

import com.twitter.escherbird.common.thriftscala.QualifiedId

import com.twitter.escherbird.metadata.thriftscala.BasicMetadata

import com.twitter.escherbird.metadata.thriftscala.EntityIndexFields

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

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.MagicFanoutCandidate

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

import com.twitter.frigate.common.base.RichEventFutCandidate

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

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

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

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

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

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

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

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

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

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

import com.twitter.topiclisting.utt.LocalizedEntity

import com.twitter.util.Future

case class FanoutReasonEntities(

userIds: Set[Long],

placeIds: Set[Long],

semanticCoreIds: Set[SemanticCoreID],

simclusterIds: Set[SimClusterID]) {

val qualifiedIds: Set[QualifiedId] =

semanticCoreIds.map(e => QualifiedId(e.domainId, e.entityId))

}

object FanoutReasonEntities {

val empty = FanoutReasonEntities(

userIds = Set.empty,

placeIds = Set.empty,

semanticCoreIds = Set.empty,

simclusterIds = Set.empty

)

def from(reasons: Seq[TargetID]): FanoutReasonEntities = {

val userIds: Set[Long] = reasons.collect {

case TargetID.UserID(userId) => userId.id

}.toSet

val placeIds: Set[Long] = reasons.collect {

case TargetID.PlaceID(placeId) => placeId.id

}.toSet

val semanticCoreIds: Set[SemanticCoreID] = reasons.collect {

case TargetID.SemanticCoreID(semanticCoreID) => semanticCoreID

}.toSet

val simclusterIds: Set[SimClusterID] = reasons.collect {

case TargetID.SimClusterID(simClusterID) => simClusterID

}.toSet

FanoutReasonEntities(

userIds = userIds,

placeIds,

semanticCoreIds = semanticCoreIds,

simclusterIds = simclusterIds

)

}

}

trait MagicFanoutHydratedCandidate extends PushCandidate with MagicFanoutCandidate {

lazy val fanoutReasonEntities: FanoutReasonEntities =

FanoutReasonEntities.from(candidateMagicEventsReasons.map(\_.reason))

}

trait MagicFanoutEventHydratedCandidate

extends MagicFanoutHydratedCandidate

with MagicFanoutEventCandidate

with RichEventFutCandidate {

def target: PushTypes.Target

def stats: StatsReceiver

def fanoutEvent: Option[FanoutEvent]

def eventFut: Future[Option[Event]]

def semanticEntityResults: Map[SemanticEntityForQuery, Option[EntityMegadata]]

def effectiveMagicEventsReasons: Option[Seq[MagicEventsReason]]

def followedTopicLocalizedEntities: Future[Set[LocalizedEntity]]

def ergLocalizedEntities: Future[Set[LocalizedEntity]]

lazy val entityAnnotationAlg: Map[TargetID, Set[AnnotationAlg]] =

fanoutEvent

.flatMap { metadata =>

metadata.eventAnnotationInfo.map { eventAnnotationInfo =>

eventAnnotationInfo.map {

case (target, annotationInfoSet) => target -> annotationInfoSet.map(\_.alg).toSet

}.toMap

}

}.getOrElse(Map.empty)

lazy val eventSource: Option[String] = fanoutEvent.map { metadata =>

val source = metadata.eventSource.getOrElse("undefined")

stats.scope("eventSource").counter(source).incr()

source

}

lazy val semanticCoreEntityTags: Map[(Long, Long), Set[String]] =

semanticEntityResults.flatMap {

case (semanticEntityForQuery, entityMegadataOpt: Option[EntityMegadata]) =>

for {

entityMegadata <- entityMegadataOpt

basicMetadata: BasicMetadata <- entityMegadata.basicMetadata

indexableFields: EntityIndexFields <- basicMetadata.indexableFields

tags <- indexableFields.tags

} yield {

((semanticEntityForQuery.domainId, semanticEntityForQuery.entityId), tags.toSet)

}

}

lazy val owningTwitterUserIds: Seq[Long] = semanticEntityResults.values.flatten

.flatMap {

\_.basicMetadata.flatMap(\_.twitter.flatMap(\_.owningTwitterUserIds))

}.flatten

.toSeq

.distinct

lazy val eventFanoutReasonEntities: FanoutReasonEntities =

fanoutEvent match {

case Some(fanout) =>

fanout.targets

.map { targets: Seq[thriftscala.Target] =>

FanoutReasonEntities.from(targets.flatMap(\_.whitelist).flatten)

}.getOrElse(FanoutReasonEntities.empty)

case \_ => FanoutReasonEntities.empty

}

override lazy val eventResultFut: Future[Event] = eventFut.map {

case Some(eventResult) => eventResult

case \_ =>

throw new IllegalArgumentException("event is None for MagicFanoutEventHydratedCandidate")

}

override val rankScore: Option[Double] = None

override val predictionScore: Option[Double] = None

}

case class MagicFanoutEventHydratedInfo(

fanoutEvent: Option[FanoutEvent],

semanticEntityResults: Map[SemanticEntityForQuery, Option[EntityMegadata]])