package com.twitter.visibility.interfaces.search

import com.twitter.decider.Decider

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.mediaservices.media\_util.GenericMediaKey

import com.twitter.servo.util.Gate

import com.twitter.stitch.Stitch

import com.twitter.strato.client.{Client => StratoClient}

import com.twitter.tweetypie.thriftscala.Tweet

import com.twitter.util.Return

import com.twitter.util.Stopwatch

import com.twitter.util.Try

import com.twitter.visibility.VisibilityLibrary

import com.twitter.visibility.builder.VerdictLogger

import com.twitter.visibility.builder.VisibilityResult

import com.twitter.visibility.builder.media.MediaFeatures

import com.twitter.visibility.builder.media.StratoMediaLabelMaps

import com.twitter.visibility.builder.tweets.\_

import com.twitter.visibility.builder.users.AuthorFeatures

import com.twitter.visibility.builder.users.RelationshipFeatures

import com.twitter.visibility.builder.users.ViewerFeatures

import com.twitter.visibility.common.MediaSafetyLabelMapSource

import com.twitter.visibility.common.MisinformationPolicySource

import com.twitter.visibility.common.SafetyLabelMapSource

import com.twitter.visibility.common.TrustedFriendsSource

import com.twitter.visibility.common.UserRelationshipSource

import com.twitter.visibility.common.UserSource

import com.twitter.visibility.rules.ComposableActions.\_

import com.twitter.visibility.configapi.configs.VisibilityDeciderGates

import com.twitter.visibility.features.FeatureMap

import com.twitter.visibility.features.TweetIsInnerQuotedTweet

import com.twitter.visibility.features.TweetIsRetweet

import com.twitter.visibility.features.TweetIsSourceTweet

import com.twitter.visibility.interfaces.common.search.SearchVFRequestContext

import com.twitter.visibility.interfaces.search.SearchVisibilityLibrary.EvaluateTweet

import com.twitter.visibility.interfaces.search.SearchVisibilityLibrary.RequestTweetId

import com.twitter.visibility.interfaces.search.TweetType.EvaluateTweetType

import com.twitter.visibility.logging.thriftscala.VFLibType

import com.twitter.visibility.models.ContentId

import com.twitter.visibility.models.ContentId.BlenderTweetId

import com.twitter.visibility.models.ContentId.TweetId

import com.twitter.visibility.models.SafetyLevel

import com.twitter.visibility.models.SafetyLevel.toThrift

import com.twitter.visibility.models.ViewerContext

import com.twitter.visibility.rules.Action

import com.twitter.visibility.rules.Allow

import com.twitter.visibility.rules.Drop

import com.twitter.visibility.rules.Interstitial

import com.twitter.visibility.rules.TweetInterstitial

object TweetType extends Enumeration {

type EvaluateTweetType = Value

val REQUEST: TweetType.Value = Value(1)

val QUOTED: TweetType.Value = Value(2)

val SOURCE: TweetType.Value = Value(3)

}

import com.twitter.visibility.interfaces.search.TweetType.\_

object SearchVisibilityLibrary {

type RequestTweetId = Long

type EvaluateTweetId = Long

type EvaluateTweet = Tweet

def buildWithStratoClient(

visibilityLibrary: VisibilityLibrary,

decider: Decider,

stratoClient: StratoClient,

userSource: UserSource,

userRelationshipSource: UserRelationshipSource

): SearchVisibilityLibrary = new SearchVisibilityLibrary(

visibilityLibrary,

decider,

stratoClient,

userSource,

userRelationshipSource,

None

)

def buildWithSafetyLabelMapSource(

visibilityLibrary: VisibilityLibrary,

decider: Decider,

stratoClient: StratoClient,

userSource: UserSource,

userRelationshipSource: UserRelationshipSource,

safetyLabelMapSource: SafetyLabelMapSource

): SearchVisibilityLibrary = new SearchVisibilityLibrary(

visibilityLibrary,

decider,

stratoClient,

userSource,

userRelationshipSource,

Some(safetyLabelMapSource)

)

def createVerdictLogger(

enableVerdictLogger: Gate[Unit],

decider: Decider,

statsReceiver: StatsReceiver

): VerdictLogger = {

if (enableVerdictLogger()) {

VerdictLogger(statsReceiver, decider)

} else {

VerdictLogger.Empty

}

}

def scribeVisibilityVerdict(

result: CombinedVisibilityResult,

enableVerdictScribing: Gate[Unit],

verdictLogger: VerdictLogger,

viewerId: Option[Long],

safetyLevel: SafetyLevel

): Unit = if (enableVerdictScribing()) {

verdictLogger.scribeVerdict(

visibilityResult = result.tweetVisibilityResult,

viewerId = viewerId,

safetyLevel = toThrift(safetyLevel),

vfLibType = VFLibType.SearchVisibilityLibrary)

result.quotedTweetVisibilityResult.map(quotedTweetVisibilityResult =>

verdictLogger.scribeVerdict(

visibilityResult = quotedTweetVisibilityResult,

viewerId = viewerId,

safetyLevel = toThrift(safetyLevel),

vfLibType = VFLibType.SearchVisibilityLibrary))

}

}

class SearchVisibilityLibrary(

visibilityLibrary: VisibilityLibrary,

decider: Decider,

stratoClient: StratoClient,

userSource: UserSource,

userRelationshipSource: UserRelationshipSource,

safetyLabelMapSourceOption: Option[SafetyLabelMapSource]) {

val libraryStatsReceiver = visibilityLibrary.statsReceiver

val stratoClientStatsReceiver = visibilityLibrary.statsReceiver.scope("strato")

val vfEngineCounter = libraryStatsReceiver.counter("vf\_engine\_requests")

val svlRequestCounter = libraryStatsReceiver.counter("svl\_requests")

val vfLatencyOverallStat = libraryStatsReceiver.stat("vf\_latency\_overall")

val vfLatencyStitchBuildStat = libraryStatsReceiver.stat("vf\_latency\_stitch\_build")

val vfLatencyStitchRunStat = libraryStatsReceiver.stat("vf\_latency\_stitch\_run")

val visibilityDeciderGates = VisibilityDeciderGates(decider)

val verdictLogger = SearchVisibilityLibrary.createVerdictLogger(

visibilityDeciderGates.enableVerdictLoggerSVL,

decider,

libraryStatsReceiver)

val tweetLabels = safetyLabelMapSourceOption match {

case Some(safetyLabelMapSource) =>

new StratoTweetLabelMaps(safetyLabelMapSource)

case None =>

new StratoTweetLabelMaps(

SafetyLabelMapSource.fromStrato(stratoClient, stratoClientStatsReceiver))

}

val mediaLabelMaps = new StratoMediaLabelMaps(

MediaSafetyLabelMapSource.fromStrato(stratoClient, stratoClientStatsReceiver))

val tweetFeatures = new TweetFeatures(tweetLabels, libraryStatsReceiver)

val searchContextFeatures = new SearchContextFeatures(libraryStatsReceiver)

val authorFeatures = new AuthorFeatures(userSource, libraryStatsReceiver)

val viewerFeatures = new ViewerFeatures(userSource, libraryStatsReceiver)

val relationshipFeatures =

new RelationshipFeatures(userRelationshipSource, libraryStatsReceiver)

val misinfoPolicySource =

MisinformationPolicySource.fromStrato(stratoClient, stratoClientStatsReceiver)

val misinfoPolicyFeatures =

new MisinformationPolicyFeatures(misinfoPolicySource, stratoClientStatsReceiver)

val exclusiveTweetFeatures =

new ExclusiveTweetFeatures(userRelationshipSource, libraryStatsReceiver)

val mediaFeatures = new MediaFeatures(mediaLabelMaps, libraryStatsReceiver)

val trustedFriendsTweetFeatures = new TrustedFriendsFeatures(

trustedFriendsSource = TrustedFriendsSource.fromStrato(stratoClient, stratoClientStatsReceiver))

val editTweetFeatures = new EditTweetFeatures(libraryStatsReceiver)

def batchProcessSearchVisibilityRequest(

batchSvRequest: BatchSearchVisibilityRequest

): Stitch[BatchSearchVisibilityResponse] = {

val elapsed = Stopwatch.start()

svlRequestCounter.incr()

val response: Stitch[BatchSearchVisibilityResponse] =

batchSvRequest.tweetContexts.groupBy(tweetContext => tweetContext.safetyLevel) map {

case (safetyLevel: SafetyLevel, tweetContexts: Seq[TweetContext]) =>

val (contentsToBeEvaluated, contentVisResultTypes) =

extractContentsToBeEvaluated(tweetContexts, batchSvRequest.viewerContext)

getVisibilityResult(

contentsToBeEvaluated,

safetyLevel,

batchSvRequest.viewerContext,

batchSvRequest.searchVFRequestContext)

.map { contentVisResults: Seq[Try[VisibilityResult]] =>

(contentVisResultTypes zip contentVisResults)

.map(handleVisibilityResultByTweetType)

.groupBy {

case (requestTweetId: RequestTweetId, (\_, \_)) => requestTweetId

}.mapValues(combineVisibilityResult)

}.onSuccess(res =>

res.values.flatten.foreach(\_ =>

SearchVisibilityLibrary.scribeVisibilityVerdict(

\_,

visibilityDeciderGates.enableVerdictScribingSVL,

verdictLogger,

batchSvRequest.viewerContext.userId,

safetyLevel)))

} reduceLeft { (left, right) =>

Stitch.joinMap(left, right)((visResultsA, visResultsB) => visResultsA ++ visResultsB)

} map { visResults =>

val (succeed, failed) = visResults.partition { case (\_, visResult) => visResult.nonEmpty }

val failedTweetIds: Seq[Long] = failed.keys.toSeq

BatchSearchVisibilityResponse(

visibilityResults = succeed.mapValues(visResult => visResult.get),

failedTweetIds = failedTweetIds

)

}

val runStitchStartMs = elapsed().inMilliseconds

val buildStitchStatMs = elapsed().inMilliseconds

vfLatencyStitchBuildStat.add(buildStitchStatMs)

response

.onSuccess(\_ => {

val overallMs = elapsed().inMilliseconds

vfLatencyOverallStat.add(overallMs)

val stitchRunMs = elapsed().inMilliseconds - runStitchStartMs

vfLatencyStitchRunStat.add(stitchRunMs)

})

}

private def extractContentsToBeEvaluated(

tweetContexts: Seq[TweetContext],

viewerContext: ViewerContext

): (

Seq[(TweetContext, EvaluateTweetType, EvaluateTweet, ContentId)],

Seq[

(RequestTweetId, EvaluateTweetType)

]

) = {

val contentsToBeEvaluated: Seq[

(TweetContext, EvaluateTweetType, EvaluateTweet, ContentId)

] = tweetContexts.map(tc =>

(

tc,

REQUEST,

tc.tweet,

getContentId(

viewerId = viewerContext.userId,

authorId = tc.tweet.coreData.get.userId,

tweet = tc.tweet))) ++

tweetContexts

.filter(tc => tc.quotedTweet.nonEmpty).map(tc =>

(

tc,

QUOTED,

tc.quotedTweet.get,

getContentId(

viewerId = viewerContext.userId,

authorId = tc.quotedTweet.get.coreData.get.userId,

tweet = tc.quotedTweet.get))) ++

tweetContexts

.filter(tc => tc.retweetSourceTweet.nonEmpty).map(tc =>

(

tc,

SOURCE,

tc.retweetSourceTweet.get,

getContentId(

viewerId = viewerContext.userId,

authorId = tc.retweetSourceTweet.get.coreData.get.userId,

tweet = tc.retweetSourceTweet.get)))

val contentVisResultTypes: Seq[(RequestTweetId, EvaluateTweetType)] = {

contentsToBeEvaluated.map {

case (tc: TweetContext, tweetType: EvaluateTweetType, \_, \_) =>

(tc.tweet.id, tweetType)

}

}

(contentsToBeEvaluated, contentVisResultTypes)

}

private def combineVisibilityResult(

visResults: Seq[(RequestTweetId, (EvaluateTweetType, Try[VisibilityResult]))]

): Option[CombinedVisibilityResult] = {

visResults.sortBy(\_.\_2.\_1)(ValueOrdering) match {

case Seq(

(\_, (REQUEST, Return(requestTweetVisResult))),

(\_, (QUOTED, Return(quotedTweetVisResult))),

(\_, (SOURCE, Return(sourceTweetVisResult)))) =>

requestTweetVisResult.verdict match {

case Allow =>

Some(CombinedVisibilityResult(sourceTweetVisResult, Some(quotedTweetVisResult)))

case \_ =>

Some(CombinedVisibilityResult(requestTweetVisResult, Some(quotedTweetVisResult)))

}

case Seq(

(\_, (REQUEST, Return(requestTweetVisResult))),

(\_, (QUOTED, Return(quotedTweetVisResult)))) =>

Some(CombinedVisibilityResult(requestTweetVisResult, Some(quotedTweetVisResult)))

case Seq(

(\_, (REQUEST, Return(requestTweetVisResult))),

(\_, (SOURCE, Return(sourceTweetVisResult)))) =>

requestTweetVisResult.verdict match {

case Allow =>

Some(CombinedVisibilityResult(sourceTweetVisResult, None))

case \_ =>

Some(CombinedVisibilityResult(requestTweetVisResult, None))

}

case Seq((\_, (REQUEST, Return(requestTweetVisResult)))) =>

Some(CombinedVisibilityResult(requestTweetVisResult, None))

case \_ => None

}

}

private def getVisibilityResult(

contents: Seq[(TweetContext, EvaluateTweetType, EvaluateTweet, ContentId)],

safetyLevel: SafetyLevel,

viewerContext: ViewerContext,

svRequestContext: SearchVFRequestContext

): Stitch[Seq[Try[VisibilityResult]]] = {

val contentContext: Map[ContentId, (TweetContext, EvaluateTweetType, EvaluateTweet)] =

contents.map {

case (

tweetContext: TweetContext,

tweetType: EvaluateTweetType,

tweet: EvaluateTweet,

contentId: ContentId) =>

contentId -> ((tweetContext, tweetType, tweet))

}.toMap

val featureMapProvider: (ContentId, SafetyLevel) => FeatureMap = {

case (contentId: ContentId, \_) =>

val (tweetContext, tweetType, tweet) = contentContext(contentId)

buildFeatureMap(

evaluatedTweet = tweet,

tweetType = tweetType,

tweetContext = tweetContext,

viewerContext = viewerContext,

svRequestContext = svRequestContext

)

}

visibilityLibrary.runRuleEngineBatch(

contentIds = contents.map { case (\_, \_, \_, id: ContentId) => id },

featureMapProvider = featureMapProvider,

viewerContext = viewerContext,

safetyLevel = safetyLevel

)

}

private def getContentId(viewerId: Option[Long], authorId: Long, tweet: Tweet): ContentId = {

if (viewerId.contains(authorId))

TweetId(tweet.id)

else BlenderTweetId(tweet.id)

}

private def buildFeatureMap(

evaluatedTweet: Tweet,

tweetType: EvaluateTweetType,

tweetContext: TweetContext,

viewerContext: ViewerContext,

svRequestContext: SearchVFRequestContext

): FeatureMap = {

val authorId = evaluatedTweet.coreData.get.userId

val viewerId = viewerContext.userId

val isRetweet =

if (tweetType.equals(REQUEST)) tweetContext.retweetSourceTweet.nonEmpty else false

val isSourceTweet = tweetType.equals(SOURCE)

val isQuotedTweet = tweetType.equals(QUOTED)

val tweetMediaKeys: Seq[GenericMediaKey] = evaluatedTweet.media

.getOrElse(Seq.empty)

.flatMap(\_.mediaKey.map(GenericMediaKey.apply))

visibilityLibrary.featureMapBuilder(

Seq(

viewerFeatures

.forViewerSearchContext(svRequestContext, viewerContext),

relationshipFeatures.forAuthorId(authorId, viewerId),

tweetFeatures.forTweet(evaluatedTweet),

mediaFeatures.forMediaKeys(tweetMediaKeys),

authorFeatures.forAuthorId(authorId),

searchContextFeatures.forSearchContext(svRequestContext),

\_.withConstantFeature(TweetIsRetweet, isRetweet),

misinfoPolicyFeatures.forTweet(evaluatedTweet, viewerContext),

exclusiveTweetFeatures.forTweet(evaluatedTweet, viewerContext),

trustedFriendsTweetFeatures.forTweet(evaluatedTweet, viewerId),

editTweetFeatures.forTweet(evaluatedTweet),

\_.withConstantFeature(TweetIsInnerQuotedTweet, isQuotedTweet),

\_.withConstantFeature(TweetIsSourceTweet, isSourceTweet),

)

)

}

private def handleVisibilityResultByTweetType(

zipVisResult: ((RequestTweetId, EvaluateTweetType), Try[VisibilityResult])

): (RequestTweetId, (EvaluateTweetType, Try[VisibilityResult])) = {

zipVisResult match {

case ((id: RequestTweetId, REQUEST), Return(visResult)) =>

(id, (REQUEST, Return(handleComposableVisibilityResult(visResult))))

case ((id: RequestTweetId, QUOTED), Return(visResult)) =>

(

id,

(

QUOTED,

Return(

handleInnerQuotedTweetVisibilityResult(handleComposableVisibilityResult(visResult)))))

case ((id: RequestTweetId, SOURCE), Return(visResult)) =>

(id, (SOURCE, Return(handleComposableVisibilityResult(visResult))))

case ((id: RequestTweetId, tweetType: EvaluateTweetType), result: Try[VisibilityResult]) =>

(id, (tweetType, result))

}

}

private def handleComposableVisibilityResult(result: VisibilityResult): VisibilityResult = {

if (result.secondaryVerdicts.nonEmpty) {

result.copy(verdict = composeActions(result.verdict, result.secondaryVerdicts))

} else {

result

}

}

private def composeActions(primary: Action, secondary: Seq[Action]): Action = {

if (primary.isComposable && secondary.nonEmpty) {

val actions = Seq[Action] { primary } ++ secondary

val interstitialOpt = Action.getFirstInterstitial(actions: \_\*)

val softInterventionOpt = Action.getFirstSoftIntervention(actions: \_\*)

val limitedEngagementsOpt = Action.getFirstLimitedEngagements(actions: \_\*)

val avoidOpt = Action.getFirstAvoid(actions: \_\*)

val numActions =

Seq[Option[\_]](interstitialOpt, softInterventionOpt, limitedEngagementsOpt, avoidOpt)

.count(\_.isDefined)

if (numActions > 1) {

TweetInterstitial(

interstitialOpt,

softInterventionOpt,

limitedEngagementsOpt,

None,

avoidOpt

)

} else {

primary

}

} else {

primary

}

}

private def handleInnerQuotedTweetVisibilityResult(

result: VisibilityResult

): VisibilityResult = {

val newVerdict: Action =

result.verdict match {

case interstitial: Interstitial => Drop(interstitial.reason)

case ComposableActionsWithInterstitial(tweetInterstitial) => Drop(tweetInterstitial.reason)

case verdict => verdict

}

result.copy(verdict = newVerdict)

}

}