package com.twitter.visibility.interfaces.tweets

import com.twitter.decider.Decider

import com.twitter.featureswitches.v2.FeatureSwitches

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.util.Stopwatch

import com.twitter.visibility.VisibilityLibrary

import com.twitter.visibility.builder.VerdictLogger

import com.twitter.visibility.builder.VisibilityResult

import com.twitter.visibility.builder.common.MutedKeywordFeatures

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

import com.twitter.visibility.builder.tweets.TweetVisibilityNudgeSourceWrapper

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.builder.users.ViewerSearchSafetyFeatures

import com.twitter.visibility.builder.users.ViewerSensitiveMediaSettingsFeatures

import com.twitter.visibility.common.\_

import com.twitter.visibility.common.actions.LimitedAction

import com.twitter.visibility.common.actions.LimitedActionType

import com.twitter.visibility.common.actions.LimitedActionsPolicy

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

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

import com.twitter.visibility.features.TweetIsInnerQuotedTweet

import com.twitter.visibility.features.TweetIsRetweet

import com.twitter.visibility.features.TweetIsSourceTweet

import com.twitter.visibility.generators.LocalizedInterstitialGenerator

import com.twitter.visibility.generators.TombstoneGenerator

import com.twitter.visibility.interfaces.tweets.enrichments.ComplianceTweetNoticeEnrichment

import com.twitter.visibility.interfaces.tweets.enrichments.LimitedActionsPolicyEnrichment

import com.twitter.visibility.interfaces.tweets.enrichments.TweetVisibilityNudgeEnrichment

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

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

import com.twitter.visibility.models.SafetyLevel

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

import com.twitter.visibility.rules.\_

object TweetVisibilityLibrary {

type Type = TweetVisibilityRequest => Stitch[VisibilityResult]

def apply(

visibilityLibrary: VisibilityLibrary,

userSource: UserSource,

userRelationshipSource: UserRelationshipSource,

keywordMatcher: KeywordMatcher.Matcher,

invitedToConversationRepo: InvitedToConversationRepo,

decider: Decider,

stratoClient: StratoClient,

localizationSource: LocalizationSource,

tweetPerspectiveSource: TweetPerspectiveSource,

tweetMediaMetadataSource: TweetMediaMetadataSource,

tombstoneGenerator: TombstoneGenerator,

interstitialGenerator: LocalizedInterstitialGenerator,

limitedActionsFeatureSwitches: FeatureSwitches,

enableParityTest: Gate[Unit] = Gate.False

): Type = {

val libraryStatsReceiver = visibilityLibrary.statsReceiver

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

val vfEngineCounter = libraryStatsReceiver.counter("vf\_engine\_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 =

createVerdictLogger(

visibilityDeciderGates.enableVerdictLoggerTVL,

decider,

libraryStatsReceiver)

val tweetLabelMaps = new StratoTweetLabelMaps(

SafetyLabelMapSource.fromStrato(stratoClient, stratoClientStatsReceiver))

val mediaLabelMaps = new StratoMediaLabelMaps(

MediaSafetyLabelMapSource.fromStrato(stratoClient, stratoClientStatsReceiver))

val tweetFeatures = new TweetFeatures(tweetLabelMaps, libraryStatsReceiver)

val tweetPerspectiveFeatures =

new TweetPerspectiveFeatures(tweetPerspectiveSource, libraryStatsReceiver)

val mediaFeatures = new MediaFeatures(mediaLabelMaps, libraryStatsReceiver)

val tweetMediaMetadataFeatures =

new TweetMediaMetadataFeatures(tweetMediaMetadataSource, libraryStatsReceiver)

val authorFeatures = new AuthorFeatures(userSource, libraryStatsReceiver)

val viewerFeatures = new ViewerFeatures(userSource, libraryStatsReceiver)

val mutedKeywordFeatures =

new MutedKeywordFeatures(

userSource,

userRelationshipSource,

keywordMatcher,

libraryStatsReceiver,

visibilityDeciderGates.enableFollowCheckInMutedKeyword

)

val relationshipFeatures =

new RelationshipFeatures(userRelationshipSource, libraryStatsReceiver)

val fonsrRelationshipFeatures =

new FosnrRelationshipFeatures(

tweetLabels = tweetLabelMaps,

userRelationshipSource = userRelationshipSource,

statsReceiver = libraryStatsReceiver)

val conversationControlFeatures =

new ConversationControlFeatures(

relationshipFeatures,

invitedToConversationRepo,

libraryStatsReceiver

)

val exclusiveTweetFeatures =

new ExclusiveTweetFeatures(userRelationshipSource, libraryStatsReceiver)

val viewerSearchSafetyFeatures = new ViewerSearchSafetyFeatures(

UserSearchSafetySource.fromStrato(stratoClient, stratoClientStatsReceiver),

libraryStatsReceiver)

val viewerSensitiveMediaSettingsFeatures = new ViewerSensitiveMediaSettingsFeatures(

UserSensitiveMediaSettingsSource.fromStrato(stratoClient, stratoClientStatsReceiver),

libraryStatsReceiver)

val toxicReplyFilterFeature = new ToxicReplyFilterFeature(statsReceiver = libraryStatsReceiver)

val misinfoPolicySource =

MisinformationPolicySource.fromStrato(stratoClient, stratoClientStatsReceiver)

val misinfoPolicyFeatures =

new MisinformationPolicyFeatures(misinfoPolicySource, stratoClientStatsReceiver)

val communityTweetFeatures = new CommunityTweetFeaturesV2(

communitiesSource = CommunitiesSource.fromStrato(

stratoClient,

stratoClientStatsReceiver

)

)

val trustedFriendsTweetFeatures = new TrustedFriendsFeatures(

trustedFriendsSource =

TrustedFriendsSource.fromStrato(stratoClient, stratoClientStatsReceiver))

val editTweetFeatures = new EditTweetFeatures(libraryStatsReceiver)

val parityTest = new TweetVisibilityLibraryParityTest(libraryStatsReceiver, stratoClient)

val localizedNudgeSource =

LocalizedNudgeSource.fromLocalizationSource(localizationSource)

val tweetVisibilityNudgeFeatures =

new TweetVisibilityNudgeSourceWrapper(localizedNudgeSource)

val localizedLimitedActionsSource =

LocalizedLimitedActionsSource.fromLocalizationSource(localizationSource)

{ r: TweetVisibilityRequest =>

val elapsed = Stopwatch.start()

var runStitchStartMs = 0L

vfEngineCounter.incr()

val contentId = TweetId(r.tweet.id)

val viewerId = r.viewerContext.userId

val authorId = coreData.userId

val tweetGenericMediaKeys = r.tweet.mediaRefs

.getOrElse(Seq.empty)

.flatMap { mediaRef =>

GenericMediaKey.fromStringKey(mediaRef.genericMediaKey)

}

val tpf =

tweetPerspectiveFeatures.forTweet(

r.tweet,

viewerId,

visibilityDeciderGates.enableFetchTweetReportedPerspective())

val featureMap =

visibilityLibrary.featureMapBuilder(

Seq(

mutedKeywordFeatures.forTweet(r.tweet, viewerId, authorId),

viewerFeatures.forViewerContext(r.viewerContext),

viewerSearchSafetyFeatures.forViewerId(viewerId),

viewerSensitiveMediaSettingsFeatures.forViewerId(viewerId),

relationshipFeatures.forAuthorId(authorId, viewerId),

fonsrRelationshipFeatures

.forTweetAndAuthorId(tweet = r.tweet, authorId = authorId, viewerId = viewerId),

tweetFeatures.forTweet(r.tweet),

tpf,

mediaFeatures.forMediaKeys(tweetGenericMediaKeys),

authorFeatures.forAuthorId(authorId),

conversationControlFeatures.forTweet(r.tweet, viewerId),

\_.withConstantFeature(TweetIsInnerQuotedTweet, r.isInnerQuotedTweet),

\_.withConstantFeature(TweetIsRetweet, r.isRetweet),

\_.withConstantFeature(TweetIsSourceTweet, r.isSourceTweet),

misinfoPolicyFeatures.forTweet(r.tweet, r.viewerContext),

exclusiveTweetFeatures.forTweet(r.tweet, r.viewerContext),

communityTweetFeatures.forTweet(r.tweet, r.viewerContext),

tweetMediaMetadataFeatures

.forTweet(

r.tweet,

tweetGenericMediaKeys,

visibilityDeciderGates.enableFetchTweetMediaMetadata()),

trustedFriendsTweetFeatures.forTweet(r.tweet, viewerId),

editTweetFeatures.forTweet(r.tweet),

toxicReplyFilterFeature.forTweet(r.tweet, viewerId),

)

)

val languageCode = r.viewerContext.requestLanguageCode.getOrElse("en")

val countryCode = r.viewerContext.requestCountryCode

val response = visibilityLibrary

.runRuleEngine(

contentId,

featureMap,

r.viewerContext,

r.safetyLevel

)

.map(

TweetVisibilityNudgeEnrichment(

\_,

tweetVisibilityNudgeFeatures,

languageCode,

countryCode))

.map(verdict => {

if (visibilityDeciderGates.enableBackendLimitedActions()) {

LimitedActionsPolicyEnrichment(

verdict,

localizedLimitedActionsSource,

languageCode,

countryCode,

limitedActionsFeatureSwitches,

libraryStatsReceiver)

} else {

verdict

}

})

.map(

handleComposableVisibilityResult(

\_,

visibilityDeciderGates.enableMediaInterstitialComposition(),

visibilityDeciderGates.enableBackendLimitedActions()))

.map(handleInnerQuotedTweetVisibilityResult(\_, r.isInnerQuotedTweet))

.map(tombstoneGenerator(\_, languageCode))

.map(interstitialGenerator(\_, languageCode))

.map(ComplianceTweetNoticeEnrichment(\_, libraryStatsReceiver))

.onSuccess(\_ => {

val overallStatMs = elapsed().inMilliseconds

vfLatencyOverallStat.add(overallStatMs)

val runStitchEndMs = elapsed().inMilliseconds

vfLatencyStitchRunStat.add(runStitchEndMs - runStitchStartMs)

})

.onSuccess(

scribeVisibilityVerdict(

\_,

visibilityDeciderGates.enableVerdictScribingTVL,

verdictLogger,

r.viewerContext.userId,

r.safetyLevel))

runStitchStartMs = elapsed().inMilliseconds

val buildStitchStatMs = elapsed().inMilliseconds

vfLatencyStitchBuildStat.add(buildStitchStatMs)

if (enableParityTest()) {

response.applyEffect { resp =>

Stitch.async(parityTest.runParityTest(r, resp))

}

} else {

response

}

}

}

def handleComposableVisibilityResult(

result: VisibilityResult,

enableMediaInterstitialComposition: Boolean,

enableBackendLimitedActions: Boolean

): VisibilityResult = {

if (result.secondaryVerdicts.nonEmpty || enableBackendLimitedActions) {

result.copy(verdict = composeActions(

result.verdict,

result.secondaryVerdicts,

enableMediaInterstitialComposition,

enableBackendLimitedActions))

} else {

result

}

}

def handleInnerQuotedTweetVisibilityResult(

result: VisibilityResult,

isInnerQuotedTweet: Boolean

): VisibilityResult = {

val newVerdict: Action =

result.verdict match {

case Interstitial(Reason.Nsfw | Reason.NsfwMedia, \_, \_) if isInnerQuotedTweet =>

Drop(Reason.Nsfw)

case ComposableActionsWithInterstitial(tweetInterstitial)

if isInnerQuotedTweet && (tweetInterstitial.reason == Reason.Nsfw || tweetInterstitial.reason == Reason.NsfwMedia) =>

Drop(Reason.Nsfw)

case verdict => verdict

}

result.copy(verdict = newVerdict)

}

def hasTweetRules(safetyLevel: SafetyLevel): Boolean = RuleBase.hasTweetRules(safetyLevel)

def composeActions(

primary: Action,

secondary: Seq[Action],

enableMediaInterstitialComposition: Boolean,

enableBackendLimitedActions: Boolean

): Action = {

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

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

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

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

val downrankOpt = Action.getFirstDownrankHomeTimeline(actions: \_\*)

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

val tweetVisibilityNudgeOpt = Action.getFirstTweetVisibilityNudge(actions: \_\*)

val mediaInterstitialOpt = {

val firstMediaInterstitialOpt = Action.getFirstMediaInterstitial(actions: \_\*)

if (enableMediaInterstitialComposition && interstitialOpt != firstMediaInterstitialOpt) {

firstMediaInterstitialOpt

} else {

None

}

}

val limitedEngagementsOpt = enableBackendLimitedActions match {

case true => buildCompositeLimitedEngagements(Action.getAllLimitedEngagements(actions: \_\*))

case false => Action.getFirstLimitedEngagements(actions: \_\*)

}

val abusiveQualityOpt = {

if (actions.contains(ConversationSectionAbusiveQuality)) {

Some(ConversationSectionAbusiveQuality)

} else {

None

}

}

val numActions =

Seq[Option[\_]](

interstitialOpt,

softInterventionOpt,

limitedEngagementsOpt,

downrankOpt,

avoidOpt,

mediaInterstitialOpt,

tweetVisibilityNudgeOpt,

abusiveQualityOpt)

.count(\_.isDefined)

if (numActions > 1) {

TweetInterstitial(

interstitialOpt,

softInterventionOpt,

limitedEngagementsOpt,

downrankOpt,

avoidOpt,

mediaInterstitialOpt,

tweetVisibilityNudgeOpt,

abusiveQualityOpt

)

} else {

if (enableBackendLimitedActions) {

limitedEngagementsOpt.getOrElse(primary)

} else {

primary

}

}

} else {

primary

}

}

def scribeVisibilityVerdict(

result: VisibilityResult,

enableVerdictScribing: Gate[Unit],

verdictLogger: VerdictLogger,

viewerId: Option[Long],

safetyLevel: SafetyLevel

): Unit = if (enableVerdictScribing()) {

verdictLogger.scribeVerdict(

visibilityResult = result,

viewerId = viewerId,

safetyLevel = toThrift(safetyLevel),

vfLibType = VFLibType.TweetVisibilityLibrary)

}

def buildCompositeLimitedEngagements(

limitedEngagements: Seq[IsLimitedEngagements]

): Option[LimitedEngagements] = {

limitedEngagements.headOption.flatMap { limitedEngagement =>

val distinctLimitedActions = limitedEngagements

.collect({ case IsLimitedEngagements(Some(policy), \_) => policy.limitedActions })

.flatten

.foldRight(Map.empty[LimitedActionType, LimitedAction])({ (limitedAction, acc) =>

acc + ((limitedAction.limitedActionType, limitedAction))

})

.values

.toSeq

if (distinctLimitedActions.nonEmpty) {

val limitedActionsPolicy = Some(LimitedActionsPolicy(distinctLimitedActions))

Some(LimitedEngagements(limitedEngagement.getLimitedEngagementReason, limitedActionsPolicy))

} else {

None

}

}

}

def createVerdictLogger(

enableVerdictLogger: Gate[Unit],

decider: Decider,

statsReceiver: StatsReceiver

): VerdictLogger = {

if (enableVerdictLogger()) {

VerdictLogger(statsReceiver, decider)

} else {

VerdictLogger.Empty

}

}

}