package com.twitter.visibility.interfaces.conversations

import com.twitter.decider.Decider

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.gizmoduck.thriftscala.User

import com.twitter.spam.rtf.thriftscala.FilteredReason

import com.twitter.spam.rtf.thriftscala.FilteredReason.UnspecifiedReason

import com.twitter.spam.rtf.thriftscala.SafetyLevel

import com.twitter.spam.rtf.thriftscala.SafetyResult

import com.twitter.stitch.Stitch

import com.twitter.timelines.render.thriftscala.RichText

import com.twitter.timelines.render.thriftscala.TombstoneDisplayType

import com.twitter.timelines.render.thriftscala.TombstoneInfo

import com.twitter.tweetypie.thriftscala.GetTweetFieldsResult

import com.twitter.tweetypie.thriftscala.TweetFieldsResultFailed

import com.twitter.tweetypie.thriftscala.TweetFieldsResultFiltered

import com.twitter.tweetypie.thriftscala.TweetFieldsResultFound

import com.twitter.tweetypie.thriftscala.TweetFieldsResultNotFound

import com.twitter.tweetypie.thriftscala.TweetFieldsResultState

import com.twitter.visibility.VisibilityLibrary

import com.twitter.visibility.builder.tweets.ModerationFeatures

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.UserRelationshipSource

import com.twitter.visibility.common.UserSource

import com.twitter.visibility.common.actions.InterstitialReason

import com.twitter.visibility.common.actions.LimitedEngagementReason

import com.twitter.visibility.common.actions.TombstoneReason

import com.twitter.visibility.common.actions.converter.scala.InterstitialReasonConverter

import com.twitter.visibility.common.actions.converter.scala.LocalizedMessageConverter

import com.twitter.visibility.common.actions.converter.scala.TombstoneReasonConverter

import com.twitter.visibility.common.filtered\_reason.FilteredReasonHelper

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

import com.twitter.visibility.features.FocalTweetId

import com.twitter.visibility.features.TweetId

import com.twitter.visibility.models.ContentId

import com.twitter.visibility.models.SafetyLevel.Tombstoning

import com.twitter.visibility.models.ViewerContext

import com.twitter.visibility.results.richtext.EpitaphToRichText

import com.twitter.visibility.results.richtext.LocalizedMessageToRichText

import com.twitter.visibility.results.urt.ReasonToUrtParser

import com.twitter.visibility.results.urt.SafetyResultToUrtParser

import com.twitter.visibility.rules.\_

import com.twitter.visibility.{thriftscala => t}

case class TombstoneVisibilityRequest(

conversationId: Long,

focalTweetId: Long,

tweets: Seq[(GetTweetFieldsResult, Option[SafetyLevel])],

authorMap: Map[

Long,

User

],

moderatedTweetIds: Seq[Long],

viewerContext: ViewerContext,

useRichText: Boolean = true)

case class TombstoneVisibilityResponse(tweetVerdicts: Map[Long, VfTombstone])

case class TombstoneVisibilityLibrary(

visibilityLibrary: VisibilityLibrary,

statsReceiver: StatsReceiver,

decider: Decider) {

private case class TombstoneType(

tweetId: Long,

tombstoneId: Long,

action: Action) {

lazy val isInnerTombstone: Boolean = tweetId != tombstoneId

lazy val tombstoneDisplayType: TombstoneDisplayType = action match {

case \_: InterstitialLimitedEngagements | \_: EmergencyDynamicInterstitial =>

TombstoneDisplayType.NonCompliant

case \_ => TombstoneDisplayType.Inline

}

}

val En: String = "en"

val View: String = "View"

val relationshipFeatures =

new RelationshipFeatures(

statsReceiver)

val visibilityDeciderGates = VisibilityDeciderGates(decider)

def toAction(

filteredReason: FilteredReason,

actionStatsReceiver: StatsReceiver

): Option[Action] = {

val enableLocalizedInterstitials =

visibilityDeciderGates.enableConvosLocalizedInterstitial()

val enableLegacyInterstitials =

visibilityDeciderGates.enableConvosLegacyInterstitial()

val tombstoneStatsReceiver = actionStatsReceiver.scope("tombstone")

val interstitialLocalStatsReceiver =

actionStatsReceiver.scope("interstitial").scope("localized")

val interstitialLegacyStatsReceiver =

actionStatsReceiver.scope("interstitial").scope("legacy")

filteredReason match {

case \_ if FilteredReasonHelper.isTombstone(filteredReason) =>

createLocalizedTombstone(filteredReason, tombstoneStatsReceiver) match {

case tombstoneOpt @ Some(LocalizedTombstone(\_, \_)) => tombstoneOpt

case \_ =>

createTombstone(Epitaph.Unavailable, tombstoneStatsReceiver, Some("emptyTombstone"))

}

case \_

if enableLocalizedInterstitials &&

FilteredReasonHelper.isLocalizedSuppressedReasonInterstitial(filteredReason) =>

FilteredReasonHelper.getLocalizedSuppressedReasonInterstitial(filteredReason) match {

case Some(t.Interstitial(reasonOpt, Some(message))) =>

InterstitialReasonConverter.fromThrift(reasonOpt).map { interstitialReason =>

interstitialLocalStatsReceiver.counter("interstitial").incr()

Interstitial(

Reason.fromInterstitialReason(interstitialReason),

Some(LocalizedMessageConverter.fromThrift(message)))

}

case \_ => None

}

case \_ if FilteredReasonHelper.containNsfwMedia(filteredReason) =>

None

case \_ if FilteredReasonHelper.possiblyUndesirable(filteredReason) =>

None

case \_ if FilteredReasonHelper.reportedTweet(filteredReason) =>

filteredReason match {

case FilteredReason.ReportedTweet(true) =>

interstitialLegacyStatsReceiver.counter("fr\_reported").incr()

Some(Interstitial(Reason.ViewerReportedAuthor))

case FilteredReason.SafetyResult(safetyResult: SafetyResult)

if enableLegacyInterstitials =>

val safetyResultReported = InterstitialReasonConverter

.fromAction(safetyResult.action).collect {

case InterstitialReason.ViewerReportedTweet => true

case InterstitialReason.ViewerReportedAuthor => true

}.getOrElse(false)

if (safetyResultReported) {

interstitialLegacyStatsReceiver.counter("reported\_author").incr()

Some(Interstitial(Reason.ViewerReportedAuthor))

} else None

case \_ => None

}

case \_ if FilteredReasonHelper.tweetMatchesViewerMutedKeyword(filteredReason) =>

filteredReason match {

case FilteredReason.TweetMatchesViewerMutedKeyword(\_) =>

interstitialLegacyStatsReceiver.counter("fr\_muted\_keyword").incr()

Some(Interstitial(Reason.MutedKeyword))

case FilteredReason.SafetyResult(safetyResult: SafetyResult)

if enableLegacyInterstitials =>

val safetyResultMutedKeyword = InterstitialReasonConverter

.fromAction(safetyResult.action).collect {

case \_: InterstitialReason.MatchesMutedKeyword => true

}.getOrElse(false)

if (safetyResultMutedKeyword) {

interstitialLegacyStatsReceiver.counter("muted\_keyword").incr()

Some(Interstitial(Reason.MutedKeyword))

} else None

case \_ => None

}

case \_ =>

None

}

}

def toAction(

tfrs: TweetFieldsResultState,

actionStatsReceiver: StatsReceiver

): Option[Action] = {

val enableLocalizedInterstitials = visibilityDeciderGates.enableConvosLocalizedInterstitial()

val enableLegacyInterstitials = visibilityDeciderGates.enableConvosLegacyInterstitial()

val tombstoneStatsReceiver = actionStatsReceiver.scope("tombstone")

val interstitialLocalStatsReceiver =

actionStatsReceiver.scope("interstitial").scope("localized")

val interstitialLegacyStatsReceiver =

actionStatsReceiver.scope("interstitial").scope("legacy")

tfrs match {

case TweetFieldsResultState.NotFound(TweetFieldsResultNotFound(\_, \_, Some(filteredReason)))

if FilteredReasonHelper.isTombstone(filteredReason) =>

createLocalizedTombstone(filteredReason, tombstoneStatsReceiver)

case TweetFieldsResultState.NotFound(tfr: TweetFieldsResultNotFound) if tfr.deleted =>

createTombstone(Epitaph.Deleted, tombstoneStatsReceiver)

case TweetFieldsResultState.NotFound(\_: TweetFieldsResultNotFound) =>

createTombstone(Epitaph.NotFound, tombstoneStatsReceiver)

case TweetFieldsResultState.Failed(TweetFieldsResultFailed(\_, \_, \_)) =>

createTombstone(Epitaph.Unavailable, tombstoneStatsReceiver, Some("failed"))

case TweetFieldsResultState.Filtered(TweetFieldsResultFiltered(UnspecifiedReason(true))) =>

createTombstone(Epitaph.Unavailable, tombstoneStatsReceiver, Some("filtered"))

case TweetFieldsResultState.Filtered(TweetFieldsResultFiltered(filteredReason)) =>

toAction(filteredReason, actionStatsReceiver)

case TweetFieldsResultState.Found(TweetFieldsResultFound(\_, \_, Some(filteredReason)))

if enableLocalizedInterstitials &&

FilteredReasonHelper.isSuppressedReasonPublicInterestInterstial(filteredReason) =>

interstitialLocalStatsReceiver.counter("ipi").incr()

FilteredReasonHelper

.getSafetyResult(filteredReason)

.flatMap(\_.reason)

.flatMap(PublicInterest.SafetyResultReasonToReason.get) match {

case Some(safetyResultReason) =>

FilteredReasonHelper

.getSuppressedReasonPublicInterestInterstial(filteredReason)

.map(edi => edi.localizedMessage)

.map(tlm => LocalizedMessageConverter.fromThrift(tlm))

.map(lm =>

InterstitialLimitedEngagements(

safetyResultReason,

Some(LimitedEngagementReason.NonCompliant),

lm))

case \_ => None

}

case TweetFieldsResultState.Found(TweetFieldsResultFound(\_, \_, Some(filteredReason)))

if enableLegacyInterstitials &&

FilteredReasonHelper.isSuppressedReasonPublicInterestInterstial(filteredReason) =>

interstitialLegacyStatsReceiver.counter("ipi").incr()

FilteredReasonHelper

.getSafetyResult(filteredReason)

.flatMap(\_.reason)

.flatMap(PublicInterest.SafetyResultReasonToReason.get)

.map(InterstitialLimitedEngagements(\_, Some(LimitedEngagementReason.NonCompliant)))

case TweetFieldsResultState.Found(TweetFieldsResultFound(\_, \_, Some(filteredReason)))

if enableLocalizedInterstitials &&

FilteredReasonHelper.isLocalizedSuppressedReasonEmergencyDynamicInterstitial(

filteredReason) =>

interstitialLocalStatsReceiver.counter("edi").incr()

FilteredReasonHelper

.getSuppressedReasonEmergencyDynamicInterstitial(filteredReason)

.map(e =>

EmergencyDynamicInterstitial(

e.copy,

e.link,

LocalizedMessageConverter.fromThrift(e.localizedMessage)))

case TweetFieldsResultState.Found(TweetFieldsResultFound(\_, \_, Some(filteredReason)))

if enableLegacyInterstitials &&

FilteredReasonHelper.isSuppressedReasonEmergencyDynamicInterstitial(filteredReason) =>

interstitialLegacyStatsReceiver.counter("edi").incr()

FilteredReasonHelper

.getSuppressedReasonEmergencyDynamicInterstitial(filteredReason)

.map(e => EmergencyDynamicInterstitial(e.copy, e.link))

case TweetFieldsResultState.Found(TweetFieldsResultFound(tweet, \_, \_))

if tweet.perspective.exists(\_.reported) =>

interstitialLegacyStatsReceiver.counter("reported").incr()

Some(Interstitial(Reason.ViewerReportedAuthor))

case TweetFieldsResultState.Found(

TweetFieldsResultFound(\_, \_, Some(UnspecifiedReason(true)))) =>

None

case TweetFieldsResultState.Found(TweetFieldsResultFound(\_, \_, Some(filteredReason))) =>

toAction(filteredReason, actionStatsReceiver)

case \_ =>

None

}

}

private[conversations] def shouldTruncateDescendantsWhenFocal(action: Action): Boolean =

action match {

case \_: InterstitialLimitedEngagements | \_: EmergencyDynamicInterstitial =>

true

case Tombstone(Epitaph.Bounced, \_) | Tombstone(Epitaph.BounceDeleted, \_) =>

true

case LocalizedTombstone(TombstoneReason.Bounced, \_) |

LocalizedTombstone(TombstoneReason.BounceDeleted, \_) =>

true

case LimitedEngagements(LimitedEngagementReason.NonCompliant, \_) =>

true

case \_ => false

}

def apply(request: TombstoneVisibilityRequest): Stitch[TombstoneVisibilityResponse] = {

val moderationFeatures = new ModerationFeatures(

moderationSource = request.moderatedTweetIds.contains,

statsReceiver = statsReceiver

)

val userSource = UserSource.fromFunction({

case (userId, \_) =>

request.authorMap

.get(userId)

.map(Stitch.value).getOrElse(Stitch.NotFound)

})

val authorFeatures = new AuthorFeatures(userSource, statsReceiver)

val viewerFeatures = new ViewerFeatures(userSource, statsReceiver)

val languageTag = request.viewerContext.requestCountryCode.getOrElse(En)

val firstRound: Seq[(GetTweetFieldsResult, Option[TombstoneType])] = request.tweets.map {

case (gtfr, safetyLevel) =>

val actionStats = statsReceiver

.scope("action")

.scope(safetyLevel.map(\_.toString().toLowerCase()).getOrElse("unknown\_safety\_level"))

toAction(gtfr.tweetResult, actionStats) match {

case Some(action) =>

(gtfr, Some(TombstoneType(gtfr.tweetId, gtfr.tweetId, action)))

case None =>

val quotedTweetId: Option[Long] = gtfr.tweetResult match {

case TweetFieldsResultState.Found(TweetFieldsResultFound(tweet, \_, \_)) =>

tweet.quotedTweet.map(\_.tweetId)

case \_ => None

}

(quotedTweetId, gtfr.quotedTweetResult) match {

case (Some(quotedTweetId), Some(tfrs)) =>

val qtActionStats = actionStats.scope("quoted")

toAction(tfrs, qtActionStats) match {

case None =>

(gtfr, None)

case Some(action) =>

(gtfr, Some(TombstoneType(gtfr.tweetId, quotedTweetId, action)))

}

case \_ =>

(gtfr, None)

}

}

}

val (firstRoundActions, secondRoundInput) = firstRound.partition {

case (\_, Some(tombstoneType)) =>

!tombstoneType.isInnerTombstone

case (\_, None) => false

}

def invokeVisibilityLibrary(tweetId: Long, author: User): Stitch[Action] = {

visibilityLibrary

.runRuleEngine(

ContentId.TweetId(tweetId),

visibilityLibrary.featureMapBuilder(

Seq(

viewerFeatures.forViewerContext(request.viewerContext),

moderationFeatures.forTweetId(tweetId),

authorFeatures.forAuthor(author),

relationshipFeatures

.forAuthor(author, request.viewerContext.userId),

\_.withConstantFeature(TweetId, tweetId),

\_.withConstantFeature(FocalTweetId, request.focalTweetId)

)

),

request.viewerContext,

Tombstoning

).map(\_.verdict)

}

val secondRoundActions: Stitch[Seq[(GetTweetFieldsResult, Option[TombstoneType])]] =

Stitch.traverse(secondRoundInput) {

case (gtfr: GetTweetFieldsResult, firstRoundTombstone: Option[TombstoneType]) =>

val secondRoundTombstone: Stitch[Option[TombstoneType]] = gtfr.tweetResult match {

case TweetFieldsResultState.Found(TweetFieldsResultFound(tweet, \_, \_)) =>

val tweetId = tweet.id

tweet.coreData

.flatMap { coreData => request.authorMap.get(coreData.userId) } match {

case Some(author) =>

invokeVisibilityLibrary(tweetId, author).flatMap {

case Allow =>

val quotedTweetId = tweet.quotedTweet.map(\_.tweetId)

val quotedTweetAuthor = tweet.quotedTweet.flatMap { qt =>

request.authorMap.get(qt.userId)

}

(quotedTweetId, quotedTweetAuthor) match {

case (Some(quotedTweetId), Some(quotedTweetAuthor)) =>

invokeVisibilityLibrary(quotedTweetId, quotedTweetAuthor).flatMap {

case Allow =>

Stitch.None

case reason =>

Stitch.value(Some(TombstoneType(tweetId, quotedTweetId, reason)))

}

case \_ =>

Stitch.None

}

case reason =>

Stitch.value(Some(TombstoneType(tweetId, tweetId, reason)))

}

case None =>

Stitch.None

}

case \_ =>

Stitch.None

}

secondRoundTombstone.map { opt => opt.orElse(firstRoundTombstone) }.map { opt =>

(gtfr, opt)

}

}

secondRoundActions.map { secondRound =>

val tombstones: Seq[(Long, VfTombstone)] = (firstRoundActions ++ secondRound).flatMap {

case (gtfr, tombstoneTypeOpt) => {

val nonCompliantLimitedEngagementsOpt = gtfr.tweetResult match {

case TweetFieldsResultState.Found(TweetFieldsResultFound(\_, \_, Some(filteredReason)))

if FilteredReasonHelper.isLimitedEngagementsNonCompliant(filteredReason) =>

Some(LimitedEngagements(LimitedEngagementReason.NonCompliant))

case \_ => None

}

(tombstoneTypeOpt, nonCompliantLimitedEngagementsOpt) match {

case (Some(tombstoneType), nonCompliantOpt) =>

val tombstoneId = tombstoneType.tombstoneId

val action = tombstoneType.action

val textOpt: Option[RichText] = action match {

case InterstitialLimitedEngagements(\_, \_, Some(localizedMessage), \_) =>

Some(LocalizedMessageToRichText(localizedMessage))

case ipi: InterstitialLimitedEngagements =>

Some(

SafetyResultToUrtParser.fromSafetyResultToRichText(

SafetyResult(

Some(PublicInterest.ReasonToSafetyResultReason(ipi.reason)),

ipi.toActionThrift()

),

languageTag

)

)

case EmergencyDynamicInterstitial(\_, \_, Some(localizedMessage), \_) =>

Some(LocalizedMessageToRichText(localizedMessage))

case edi: EmergencyDynamicInterstitial =>

Some(

SafetyResultToUrtParser.fromSafetyResultToRichText(

SafetyResult(

None,

edi.toActionThrift()

),

languageTag

)

)

case Tombstone(epitaph, \_) =>

if (request.useRichText)

Some(EpitaphToRichText(epitaph, languageTag))

else

Some(EpitaphToRichText(Epitaph.UnavailableWithoutLink, languageTag))

case LocalizedTombstone(\_, message) =>

if (request.useRichText)

Some(LocalizedMessageToRichText(LocalizedMessageConverter.toThrift(message)))

else

Some(EpitaphToRichText(Epitaph.UnavailableWithoutLink, languageTag))

case Interstitial(\_, Some(localizedMessage), \_) =>

Some(LocalizedMessageToRichText.apply(localizedMessage))

case interstitial: Interstitial =>

ReasonToUrtParser.fromReasonToRichText(interstitial.reason, languageTag)

case \_ =>

None

}

val isRoot: Boolean = gtfr.tweetId == request.conversationId

val isOuter: Boolean = tombstoneId == request.conversationId

val revealTextOpt: Option[RichText] = action match {

case \_: InterstitialLimitedEngagements | \_: EmergencyDynamicInterstitial

if isRoot && isOuter =>

None

case \_: Interstitial | \_: InterstitialLimitedEngagements |

\_: EmergencyDynamicInterstitial =>

Some(ReasonToUrtParser.getRichRevealText(languageTag))

case \_ =>

None

}

val includeTweet = action match {

case \_: Interstitial | \_: InterstitialLimitedEngagements |

\_: EmergencyDynamicInterstitial =>

true

case \_ => false

}

val truncateForAction: Boolean =

shouldTruncateDescendantsWhenFocal(action)

val truncateForNonCompliant: Boolean =

nonCompliantOpt

.map(shouldTruncateDescendantsWhenFocal).getOrElse(false)

val truncateDescendants: Boolean =

truncateForAction || truncateForNonCompliant

val tombstone = textOpt match {

case Some(\_) if request.useRichText =>

VfTombstone(

includeTweet = includeTweet,

action = action,

tombstoneInfo = Some(

TombstoneInfo(

cta = None,

revealText = None,

richText = textOpt,

richRevealText = revealTextOpt

)

),

tombstoneDisplayType = tombstoneType.tombstoneDisplayType,

truncateDescendantsWhenFocal = truncateDescendants

)

case Some(\_) =>

VfTombstone(

includeTweet = includeTweet,

action = action,

tombstoneInfo = Some(

TombstoneInfo(

text = textOpt

.map(richText => richText.text).getOrElse(

""

cta = None,

revealText = revealTextOpt.map(\_.text),

richText = None,

richRevealText = None

)

),

tombstoneDisplayType = tombstoneType.tombstoneDisplayType,

truncateDescendantsWhenFocal = truncateDescendants

)

case None =>

VfTombstone(

includeTweet = false,

action = action,

tombstoneInfo = Some(

TombstoneInfo(

cta = None,

revealText = None,

richText = Some(EpitaphToRichText(Epitaph.Unavailable, languageTag)),

richRevealText = None

)

),

tombstoneDisplayType = tombstoneType.tombstoneDisplayType,

truncateDescendantsWhenFocal = truncateDescendants

)

}

Some((gtfr.tweetId, tombstone))

case (None, Some(limitedEngagements))

if shouldTruncateDescendantsWhenFocal(limitedEngagements) =>

val tombstone = VfTombstone(

tombstoneId = gtfr.tweetId,

includeTweet = true,

action = limitedEngagements,

tombstoneInfo = None,

tombstoneDisplayType = TombstoneDisplayType.NonCompliant,

truncateDescendantsWhenFocal = true

)

Some((gtfr.tweetId, tombstone))

case \_ =>

None

}

}

}

TombstoneVisibilityResponse(

tweetVerdicts = tombstones.toMap

)

}

}

private def createLocalizedTombstone(

filteredReason: FilteredReason,

tombstoneStats: StatsReceiver,

): Option[LocalizedTombstone] = {

val tombstoneOpt = FilteredReasonHelper.getTombstone(filteredReason)

tombstoneOpt match {

case Some(t.Tombstone(reasonOpt, Some(message))) =>

TombstoneReasonConverter.fromThrift(reasonOpt).map { localReason =>

tombstoneStats

.scope("localized").counter(localReason.toString().toLowerCase()).incr()

LocalizedTombstone(localReason, LocalizedMessageConverter.fromThrift(message))

}

case \_ => None

}

}

private def createTombstone(

epitaph: Epitaph,

tombstoneStats: StatsReceiver,

extraCounterOpt: Option[String] = None

): Option[Action] = {

tombstoneStats

.scope("legacy")

.counter(epitaph.toString().toLowerCase())

.incr()

extraCounterOpt.map { extraCounter =>

tombstoneStats

.scope("legacy")

.scope(epitaph.toString().toLowerCase())

.counter(extraCounter)

.incr()

}

Some(Tombstone(epitaph))

}

}