package com.twitter.product\_mixer.component\_library.candidate\_source.tweetconvosvc

import com.twitter.product\_mixer.core.feature.featuremap.FeatureMap

import com.twitter.product\_mixer.core.functional\_component.candidate\_source.CandidateSourceWithExtractedFeatures

import com.twitter.product\_mixer.core.functional\_component.candidate\_source.CandidatesWithSourceFeatures

import com.twitter.product\_mixer.core.model.common.identifier.CandidateSourceIdentifier

import com.twitter.stitch.Stitch

import com.twitter.tweetconvosvc.tweet\_ancestor.{thriftscala => ta}

import com.twitter.tweetconvosvc.{thriftscala => tcs}

import com.twitter.util.Return

import com.twitter.util.Throw

import javax.inject.Inject

import javax.inject.Singleton

case class ConversationServiceCandidateSourceRequest(

tweetsWithConversationMetadata: Seq[TweetWithConversationMetadata])

case class TweetWithConversationMetadata(

tweetId: Long,

userId: Option[Long],

sourceTweetId: Option[Long],

sourceUserId: Option[Long],

inReplyToTweetId: Option[Long],

conversationId: Option[Long],

ancestors: Seq[ta.TweetAncestor])

/\*\*

\* Candidate source that fetches ancestors of input candidates from Tweetconvosvc and

\* returns a flattened list of input and ancestor candidates.

\*/

@Singleton

class ConversationServiceCandidateSource @Inject() (

conversationServiceClient: tcs.ConversationService.MethodPerEndpoint)

extends CandidateSourceWithExtractedFeatures[

ConversationServiceCandidateSourceRequest,

TweetWithConversationMetadata

] {

override val identifier: CandidateSourceIdentifier =

CandidateSourceIdentifier("ConversationService")

private val maxModuleSize = 3

private val maxAncestorsInConversation = 2

private val numberOfRootTweets = 1

private val maxTweetsInConversationWithSameId = 1

override def apply(

request: ConversationServiceCandidateSourceRequest

): Stitch[CandidatesWithSourceFeatures[TweetWithConversationMetadata]] = {

val inputTweetsWithConversationMetadata: Seq[TweetWithConversationMetadata] =

request.tweetsWithConversationMetadata

val ancestorsRequest =

tcs.GetAncestorsRequest(inputTweetsWithConversationMetadata.map(\_.tweetId))

// build the tweets with conversation metadata by calling the conversation service with reduced

// ancestors to limit to maxModuleSize

val tweetsWithConversationMetadataFromAncestors: Stitch[Seq[TweetWithConversationMetadata]] =

Stitch

.callFuture(conversationServiceClient.getAncestors(ancestorsRequest))

.map { getAncestorsResponse: tcs.GetAncestorsResponse =>

inputTweetsWithConversationMetadata

.zip(getAncestorsResponse.ancestors).collect {

case (focalTweet, tcs.TweetAncestorsResult.TweetAncestors(ancestorsResult))

if ancestorsResult.nonEmpty =>

getTweetsInThread(focalTweet, ancestorsResult.head)

}.flatten

}

// dedupe the tweets in the list and transform the calling error to

// return the requested tweets with conversation metadata

val transformedTweetsWithConversationMetadata: Stitch[Seq[TweetWithConversationMetadata]] =

tweetsWithConversationMetadataFromAncestors.transform {

case Return(ancestors) =>

Stitch.value(dedupeCandidates(inputTweetsWithConversationMetadata, ancestors))

case Throw(\_) =>

Stitch.value(inputTweetsWithConversationMetadata)

}

// return the candidates with empty source features from transformed tweetsWithConversationMetadata

transformedTweetsWithConversationMetadata.map {

responseTweetsWithConversationMetadata: Seq[TweetWithConversationMetadata] =>

CandidatesWithSourceFeatures(

responseTweetsWithConversationMetadata,

FeatureMap.empty

)

}

}

private def getTweetsInThread(

focalTweet: TweetWithConversationMetadata,

ancestors: ta.TweetAncestors

): Seq[TweetWithConversationMetadata] = {

// Re-add the focal tweet so we can easily build modules and dedupe later.

// Note, TweetConvoSVC returns the bottom of the thread first, so we

// reverse them for easy rendering.

val focalTweetWithConversationMetadata = TweetWithConversationMetadata(

tweetId = focalTweet.tweetId,

userId = focalTweet.userId,

sourceTweetId = focalTweet.sourceTweetId,

sourceUserId = focalTweet.sourceUserId,

inReplyToTweetId = focalTweet.inReplyToTweetId,

conversationId = Some(focalTweet.tweetId),

ancestors = ancestors.ancestors

)

val parentTweets = ancestors.ancestors.map { ancestor =>

TweetWithConversationMetadata(

tweetId = ancestor.tweetId,

userId = Some(ancestor.userId),

sourceTweetId = None,

sourceUserId = None,

inReplyToTweetId = None,

conversationId = Some(focalTweet.tweetId),

ancestors = Seq.empty

)

} ++ getTruncatedRootTweet(ancestors, focalTweet.tweetId)

val (intermediates, root) = parentTweets.splitAt(parentTweets.size - numberOfRootTweets)

val truncatedIntermediates =

intermediates.take(maxModuleSize - maxAncestorsInConversation).reverse

root ++ truncatedIntermediates :+ focalTweetWithConversationMetadata

}

/\*\*

\* Ancestor store truncates at 256 ancestors. For very large reply threads, we try best effort

\* to append the root tweet to the ancestor list based on the conversationId and

\* conversationRootAuthorId. When rendering conversation modules, we can display the root tweet

\* instead of the 256th highest ancestor.

\*/

private def getTruncatedRootTweet(

ancestors: ta.TweetAncestors,

focalTweetId: Long

): Option[TweetWithConversationMetadata] = {

ancestors.conversationRootAuthorId.collect {

case rootAuthorId

if ancestors.state == ta.ReplyState.Partial &&

ancestors.ancestors.last.tweetId != ancestors.conversationId =>

TweetWithConversationMetadata(

tweetId = ancestors.conversationId,

userId = Some(rootAuthorId),

sourceTweetId = None,

sourceUserId = None,

inReplyToTweetId = None,

conversationId = Some(focalTweetId),

ancestors = Seq.empty

)

}

}

private def dedupeCandidates(

inputTweetsWithConversationMetadata: Seq[TweetWithConversationMetadata],

ancestors: Seq[TweetWithConversationMetadata]

): Seq[TweetWithConversationMetadata] = {

val dedupedAncestors: Iterable[TweetWithConversationMetadata] = ancestors

.groupBy(\_.tweetId).map {

case (\_, duplicateAncestors)

if duplicateAncestors.size > maxTweetsInConversationWithSameId =>

duplicateAncestors.maxBy(\_.conversationId.getOrElse(0L))

case (\_, nonDuplicateAncestors) => nonDuplicateAncestors.head

}

// Sort by tweet id to prevent issues with future assumptions of the root being the first

// tweet and the focal being the last tweet in a module. The tweets as a whole do not need

// to be sorted overall, only the relative order within modules must be kept.

val sortedDedupedAncestors: Seq[TweetWithConversationMetadata] =

dedupedAncestors.toSeq.sortBy(\_.tweetId)

val ancestorIds = sortedDedupedAncestors.map(\_.tweetId).toSet

val updatedCandidates = inputTweetsWithConversationMetadata.filterNot { candidate =>

ancestorIds.contains(candidate.tweetId)

}

sortedDedupedAncestors ++ updatedCandidates

}

}