package com.twitter.tsp.stores

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.FailureFlags.flagsOf

import com.twitter.finagle.mux.ClientDiscardedRequestException

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.store.interests

import com.twitter.simclusters\_v2.common.UserId

import com.twitter.storehaus.ReadableStore

import com.twitter.topiclisting.ProductId

import com.twitter.topiclisting.TopicListing

import com.twitter.topiclisting.TopicListingViewerContext

import com.twitter.topiclisting.{SemanticCoreEntityId => ScEntityId}

import com.twitter.tsp.thriftscala.TopicFollowType

import com.twitter.tsp.thriftscala.TopicListingSetting

import com.twitter.tsp.thriftscala.TopicSocialProofFilteringBypassMode

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.TimeoutException

import com.twitter.util.Timer

class UttTopicFilterStore(

topicListing: TopicListing,

userOptOutTopicsStore: ReadableStore[interests.UserId, TopicResponses],

explicitFollowingTopicsStore: ReadableStore[interests.UserId, TopicResponses],

notInterestedTopicsStore: ReadableStore[interests.UserId, TopicResponses],

localizedUttRecommendableTopicsStore: ReadableStore[LocalizedUttTopicNameRequest, Set[Long]],

timer: Timer,

stats: StatsReceiver) {

import UttTopicFilterStore.\_

// Set of blacklisted SemanticCore IDs that are paused.

private[this] def getPausedTopics(topicCtx: TopicListingViewerContext): Set[ScEntityId] = {

topicListing.getPausedTopics(topicCtx)

}

private[this] def getOptOutTopics(userId: Long): Future[Set[ScEntityId]] = {

stats.counter("getOptOutTopicsCount").incr()

userOptOutTopicsStore

.get(userId).map { responseOpt =>

responseOpt

.map { responses => responses.responses.map(\_.entityId) }.getOrElse(Seq.empty).toSet

}.raiseWithin(DefaultOptOutTimeout)(timer).rescue {

case err: TimeoutException =>

stats.counter("getOptOutTopicsTimeout").incr()

Future.exception(err)

case err: ClientDiscardedRequestException

if flagsOf(err).contains("interrupted") && flagsOf(err)

.contains("ignorable") =>

stats.counter("getOptOutTopicsDiscardedBackupRequest").incr()

Future.exception(err)

case err =>

stats.counter("getOptOutTopicsFailure").incr()

Future.exception(err)

}

}

private[this] def getNotInterestedIn(userId: Long): Future[Set[ScEntityId]] = {

stats.counter("getNotInterestedInCount").incr()

notInterestedTopicsStore

.get(userId).map { responseOpt =>

responseOpt

.map { responses => responses.responses.map(\_.entityId) }.getOrElse(Seq.empty).toSet

}.raiseWithin(DefaultNotInterestedInTimeout)(timer).rescue {

case err: TimeoutException =>

stats.counter("getNotInterestedInTimeout").incr()

Future.exception(err)

case err: ClientDiscardedRequestException

if flagsOf(err).contains("interrupted") && flagsOf(err)

.contains("ignorable") =>

stats.counter("getNotInterestedInDiscardedBackupRequest").incr()

Future.exception(err)

case err =>

stats.counter("getNotInterestedInFailure").incr()

Future.exception(err)

}

}

private[this] def getFollowedTopics(userId: Long): Future[Set[TopicResponse]] = {

stats.counter("getFollowedTopicsCount").incr()

explicitFollowingTopicsStore

.get(userId).map { responseOpt =>

responseOpt.map(\_.responses.toSet).getOrElse(Set.empty)

}.raiseWithin(DefaultInterestedInTimeout)(timer).rescue {

case \_: TimeoutException =>

stats.counter("getFollowedTopicsTimeout").incr()

Future(Set.empty)

case \_ =>

stats.counter("getFollowedTopicsFailure").incr()

Future(Set.empty)

}

}

private[this] def getFollowedTopicIds(userId: Long): Future[Set[ScEntityId]] = {

getFollowedTopics(userId: Long).map(\_.map(\_.entityId))

}

private[this] def getWhitelistTopicIds(

normalizedContext: TopicListingViewerContext,

enableInternationalTopics: Boolean

): Future[Set[ScEntityId]] = {

stats.counter("getWhitelistTopicIdsCount").incr()

val uttRequest = LocalizedUttTopicNameRequest(

productId = ProductId.Followable,

viewerContext = normalizedContext,

enableInternationalTopics = enableInternationalTopics

)

localizedUttRecommendableTopicsStore

.get(uttRequest).map { response =>

response.getOrElse(Set.empty)

}.rescue {

case \_ =>

stats.counter("getWhitelistTopicIdsFailure").incr()

Future(Set.empty)

}

}

private[this] def getDenyListTopicIdsForUser(

userId: UserId,

topicListingSetting: TopicListingSetting,

context: TopicListingViewerContext,

bypassModes: Option[Set[TopicSocialProofFilteringBypassMode]]

): Future[Set[ScEntityId]] = {

val denyListTopicIdsFuture = topicListingSetting match {

case TopicListingSetting.ImplicitFollow =>

getFollowedTopicIds(userId)

case \_ =>

Future(Set.empty[ScEntityId])

}

// we don't filter opt-out topics for implicit follow topic listing setting

val optOutTopicIdsFuture = topicListingSetting match {

case TopicListingSetting.ImplicitFollow => Future(Set.empty[ScEntityId])

case \_ => getOptOutTopics(userId)

}

val notInterestedTopicIdsFuture =

if (bypassModes.exists(\_.contains(TopicSocialProofFilteringBypassMode.NotInterested))) {

Future(Set.empty[ScEntityId])

} else {

getNotInterestedIn(userId)

}

val pausedTopicIdsFuture = Future.value(getPausedTopics(context))

Future

.collect(

List(

denyListTopicIdsFuture,

optOutTopicIdsFuture,

notInterestedTopicIdsFuture,

pausedTopicIdsFuture)).map { list => list.reduce(\_ ++ \_) }

}

private[this] def getDiff(

aFut: Future[Set[ScEntityId]],

bFut: Future[Set[ScEntityId]]

): Future[Set[ScEntityId]] = {

Future.join(aFut, bFut).map {

case (a, b) => a.diff(b)

}

}

/\*\*

\* calculates the diff of all the whitelisted IDs with blacklisted IDs and returns the set of IDs

\* that we will be recommending from or followed topics by the user by client setting.

\*/

def getAllowListTopicsForUser(

userId: UserId,

topicListingSetting: TopicListingSetting,

context: TopicListingViewerContext,

bypassModes: Option[Set[TopicSocialProofFilteringBypassMode]]

): Future[Map[ScEntityId, Option[TopicFollowType]]] = {

/\*\*

\* Title: an illustrative table to explain how allow list is composed

\* AllowList = WhiteList - DenyList - OptOutTopics - PausedTopics - NotInterestedInTopics

\*

\* TopicListingSetting: Following ImplicitFollow All Followable

\* Whitelist: FollowedTopics(user) AllWhitelistedTopics Nil AllWhitelistedTopics

\* DenyList: Nil FollowedTopics(user) Nil Nil

\*

\* ps. for TopicListingSetting.All, the returned allow list is Nil. Why?

\* It's because that allowList is not required given the TopicListingSetting == 'All'.

\* See TopicSocialProofHandler.filterByAllowedList() for more details.

\*/

topicListingSetting match {

// "All" means all the UTT entity is qualified. So don't need to fetch the Whitelist anymore.

case TopicListingSetting.All => Future.value(Map.empty)

case TopicListingSetting.Following =>

getFollowingTopicsForUserWithTimestamp(userId, context, bypassModes).map {

\_.mapValues(\_ => Some(TopicFollowType.Following))

}

case TopicListingSetting.ImplicitFollow =>

getDiff(

getWhitelistTopicIds(context, enableInternationalTopics = true),

getDenyListTopicIdsForUser(userId, topicListingSetting, context, bypassModes)).map {

\_.map { scEntityId =>

scEntityId -> Some(TopicFollowType.ImplicitFollow)

}.toMap

}

case \_ =>

val followedTopicIdsFut = getFollowedTopicIds(userId)

val allowListTopicIdsFut = getDiff(

getWhitelistTopicIds(context, enableInternationalTopics = true),

getDenyListTopicIdsForUser(userId, topicListingSetting, context, bypassModes))

Future.join(allowListTopicIdsFut, followedTopicIdsFut).map {

case (allowListTopicId, followedTopicIds) =>

allowListTopicId.map { scEntityId =>

if (followedTopicIds.contains(scEntityId))

scEntityId -> Some(TopicFollowType.Following)

else scEntityId -> Some(TopicFollowType.ImplicitFollow)

}.toMap

}

}

}

private[this] def getFollowingTopicsForUserWithTimestamp(

userId: UserId,

context: TopicListingViewerContext,

bypassModes: Option[Set[TopicSocialProofFilteringBypassMode]]

): Future[Map[ScEntityId, Option[Long]]] = {

val followedTopicIdToTimestampFut = getFollowedTopics(userId).map(\_.map { followedTopic =>

followedTopic.entityId -> followedTopic.topicFollowTimestamp

}.toMap)

followedTopicIdToTimestampFut.flatMap { followedTopicIdToTimestamp =>

getDiff(

Future(followedTopicIdToTimestamp.keySet),

getDenyListTopicIdsForUser(userId, TopicListingSetting.Following, context, bypassModes)

).map {

\_.map { scEntityId =>

scEntityId -> followedTopicIdToTimestamp.get(scEntityId).flatten

}.toMap

}

}

}

}

object UttTopicFilterStore {

val DefaultNotInterestedInTimeout: Duration = 60.milliseconds

val DefaultOptOutTimeout: Duration = 60.milliseconds

val DefaultInterestedInTimeout: Duration = 60.milliseconds

}