package com.twitter.usersignalservice.base

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.Stats

import com.twitter.storehaus.ReadableStore

import com.twitter.usersignalservice.thriftscala.Signal

import com.twitter.usersignalservice.thriftscala.SignalType

import com.twitter.util.Future

import com.twitter.util.Timer

/\*\*

\* Combine a BaseSignalFetcher with a map of negative signalFetchers. Filter out the negative

\* signals from the signals from BaseSignalFetcher.

\*/

case class FilteredSignalFetcherController(

backingSignalFetcher: BaseSignalFetcher,

originSignalType: SignalType,

stats: StatsReceiver,

timer: Timer,

filterSignalFetchers: Map[SignalType, BaseSignalFetcher] =

Map.empty[SignalType, BaseSignalFetcher])

extends ReadableStore[Query, Seq[Signal]] {

val statsReceiver: StatsReceiver = stats.scope(this.getClass.getCanonicalName)

override def get(query: Query): Future[Option[Seq[Signal]]] = {

val clientStatsReceiver = statsReceiver.scope(query.signalType.name).scope(query.clientId.name)

Stats

.trackItems(clientStatsReceiver) {

val backingSignals =

backingSignalFetcher.get(Query(query.userId, originSignalType, None, query.clientId))

val filteredSignals = filter(query, backingSignals)

filteredSignals

}.raiseWithin(BaseSignalFetcher.Timeout)(timer).handle {

case e =>

clientStatsReceiver.scope("FetcherExceptions").counter(e.getClass.getCanonicalName).incr()

BaseSignalFetcher.EmptyResponse

}

}

def filter(

query: Query,

rawSignals: Future[Option[Seq[Signal]]]

): Future[Option[Seq[Signal]]] = {

Stats

.trackItems(statsReceiver) {

val originSignals = rawSignals.map(\_.getOrElse(Seq.empty[Signal]))

val filterSignals =

Future

.collect {

filterSignalFetchers.map {

case (signalType, signalFetcher) =>

signalFetcher

.get(Query(query.userId, signalType, None, query.clientId))

.map(\_.getOrElse(Seq.empty))

}.toSeq

}.map(\_.flatten.toSet)

val filterSignalsSet = filterSignals

.map(\_.flatMap(\_.targetInternalId))

val originSignalsWithId =

originSignals.map(\_.map(signal => (signal, signal.targetInternalId)))

Future.join(originSignalsWithId, filterSignalsSet).map {

case (originSignalsWithId, filterSignalsSet) =>

Some(

originSignalsWithId

.collect {

case (signal, internalIdOpt)

if internalIdOpt.nonEmpty && !filterSignalsSet.contains(internalIdOpt.get) =>

signal

}.take(query.maxResults.getOrElse(Int.MaxValue)))

}

}

}

}