package com.twitter.cr\_mixer.source\_signal

import com.twitter.cr\_mixer.config.TimeoutConfig

import com.twitter.cr\_mixer.model.ModuleNames

import com.twitter.cr\_mixer.model.SourceInfo

import com.twitter.cr\_mixer.thriftscala.SourceType

import com.twitter.cr\_mixer.source\_signal.SourceFetcher.FetcherQuery

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.storehaus.ReadableStore

import com.twitter.usersignalservice.thriftscala.{Signal => UssSignal}

import com.twitter.usersignalservice.thriftscala.SignalType

import com.twitter.frigate.common.util.StatsUtil.Size

import com.twitter.frigate.common.util.StatsUtil.Success

import com.twitter.frigate.common.util.StatsUtil.Empty

import com.twitter.util.Future

import com.twitter.util.Time

import javax.inject.Singleton

import javax.inject.Inject

import javax.inject.Named

@Singleton

case class UssSourceSignalFetcher @Inject() (

@Named(ModuleNames.UssStore) ussStore: ReadableStore[UssStore.Query, Seq[

(SignalType, Seq[UssSignal])

]],

override val timeoutConfig: TimeoutConfig,

globalStats: StatsReceiver)

extends SourceSignalFetcher {

override protected val stats: StatsReceiver = globalStats.scope(identifier)

override type SignalConvertType = UssSignal

// always enable USS call. We have fine-grained FS to decider which signal to fetch

override def isEnabled(query: FetcherQuery): Boolean = true

override def fetchAndProcess(

query: FetcherQuery,

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

// Fetch raw signals

val rawSignals = ussStore.get(UssStore.Query(query.userId, query.params, query.product)).map {

\_.map {

\_.map {

case (signalType, signals) =>

trackUssSignalStatsPerSignalType(query, signalType, signals)

(signalType, signals)

}

}

}

/\*\*

\* Process signals:

\* Transform a Seq of USS Signals with signalType specified to a Seq of SourceInfo

\* We do case match to make sure the SignalType can correctly map to a SourceType defined in CrMixer

\* and it should be simplified.

\*/

rawSignals.map {

\_.map { nestedSignal =>

val sourceInfoList = nestedSignal.flatMap {

case (signalType, ussSignals) =>

signalType match {

case SignalType.TweetFavorite =>

convertSourceInfo(sourceType = SourceType.TweetFavorite, signals = ussSignals)

case SignalType.Retweet =>

convertSourceInfo(sourceType = SourceType.Retweet, signals = ussSignals)

case SignalType.Reply =>

convertSourceInfo(sourceType = SourceType.Reply, signals = ussSignals)

case SignalType.OriginalTweet =>

convertSourceInfo(sourceType = SourceType.OriginalTweet, signals = ussSignals)

case SignalType.AccountFollow =>

convertSourceInfo(sourceType = SourceType.UserFollow, signals = ussSignals)

case SignalType.RepeatedProfileVisit180dMinVisit6V1 |

SignalType.RepeatedProfileVisit90dMinVisit6V1 |

SignalType.RepeatedProfileVisit14dMinVisit2V1 =>

convertSourceInfo(

sourceType = SourceType.UserRepeatedProfileVisit,

signals = ussSignals)

case SignalType.NotificationOpenAndClickV1 =>

convertSourceInfo(sourceType = SourceType.NotificationClick, signals = ussSignals)

case SignalType.TweetShareV1 =>

convertSourceInfo(sourceType = SourceType.TweetShare, signals = ussSignals)

case SignalType.RealGraphOon =>

convertSourceInfo(sourceType = SourceType.RealGraphOon, signals = ussSignals)

case SignalType.GoodTweetClick | SignalType.GoodTweetClick5s |

SignalType.GoodTweetClick10s | SignalType.GoodTweetClick30s =>

convertSourceInfo(sourceType = SourceType.GoodTweetClick, signals = ussSignals)

case SignalType.VideoView90dPlayback50V1 =>

convertSourceInfo(

sourceType = SourceType.VideoTweetPlayback50,

signals = ussSignals)

case SignalType.VideoView90dQualityV1 =>

convertSourceInfo(

sourceType = SourceType.VideoTweetQualityView,

signals = ussSignals)

case SignalType.GoodProfileClick | SignalType.GoodProfileClick20s |

SignalType.GoodProfileClick30s =>

convertSourceInfo(sourceType = SourceType.GoodProfileClick, signals = ussSignals)

// negative signals

case SignalType.AccountBlock =>

convertSourceInfo(sourceType = SourceType.AccountBlock, signals = ussSignals)

case SignalType.AccountMute =>

convertSourceInfo(sourceType = SourceType.AccountMute, signals = ussSignals)

case SignalType.TweetReport =>

convertSourceInfo(sourceType = SourceType.TweetReport, signals = ussSignals)

case SignalType.TweetDontLike =>

convertSourceInfo(sourceType = SourceType.TweetDontLike, signals = ussSignals)

// Aggregated Signals

case SignalType.TweetBasedUnifiedEngagementWeightedSignal |

SignalType.TweetBasedUnifiedUniformSignal =>

convertSourceInfo(sourceType = SourceType.TweetAggregation, signals = ussSignals)

case SignalType.ProducerBasedUnifiedEngagementWeightedSignal |

SignalType.ProducerBasedUnifiedUniformSignal =>

convertSourceInfo(sourceType = SourceType.ProducerAggregation, signals = ussSignals)

// Default

case \_ =>

Seq.empty[SourceInfo]

}

}

sourceInfoList

}

}

}

override def convertSourceInfo(

sourceType: SourceType,

signals: Seq[SignalConvertType]

): Seq[SourceInfo] = {

signals.map { signal =>

SourceInfo(

sourceType = sourceType,

internalId = signal.targetInternalId.getOrElse(

throw new IllegalArgumentException(

s"${sourceType.toString} Signal does not have internalId")),

sourceEventTime =

if (signal.timestamp == 0L) None else Some(Time.fromMilliseconds(signal.timestamp))

)

}

}

private def trackUssSignalStatsPerSignalType(

query: FetcherQuery,

signalType: SignalType,

ussSignals: Seq[UssSignal]

): Unit = {

val productScopedStats = stats.scope(query.product.originalName)

val productUserStateScopedStats = productScopedStats.scope(query.userState.toString)

val productStats = productScopedStats.scope(signalType.toString)

val productUserStateStats = productUserStateScopedStats.scope(signalType.toString)

productStats.counter(Success).incr()

productUserStateStats.counter(Success).incr()

val size = ussSignals.size

productStats.stat(Size).add(size)

productUserStateStats.stat(Size).add(size)

if (size == 0) {

productStats.counter(Empty).incr()

productUserStateStats.counter(Empty).incr()

}

}

}