package com.twitter.usersignalservice

package base

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.storehaus.ReadableStore

import com.twitter.usersignalservice.thriftscala.Signal

import com.twitter.util.Future

import com.twitter.twistly.common.UserId

import com.twitter.usersignalservice.thriftscala.SignalType

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

import com.twitter.conversions.DurationOps.\_

import com.twitter.usersignalservice.thriftscala.ClientIdentifier

import com.twitter.util.Duration

import com.twitter.util.Timer

import java.io.Serializable

case class Query(

userId: UserId,

signalType: SignalType,

maxResults: Option[Int],

clientId: ClientIdentifier = ClientIdentifier.Unknown)

/\*\*

\* A trait that defines a standard interface for the signal fetcher

\*

\* Extends this only when all other traits extending BaseSignalFetcher do not apply to

\* your use case.

\*/

trait BaseSignalFetcher extends ReadableStore[Query, Seq[Signal]] {

import BaseSignalFetcher.\_

/\*\*

\* This RawSignalType is the output type of `getRawSignals` and the input type of `process`.

\* Override it as your own raw signal type to maintain meta data which can be used in the

\* step of `process`.

\* Note that the RawSignalType is an intermediate data type intended to be small to avoid

\* big data chunks being passed over functions or being memcached.

\*/

type RawSignalType <: Serializable

def name: String

def statsReceiver: StatsReceiver

def timer: Timer

/\*\*

\* This function is called by the top level class to fetch signals. It executes the pipeline to

\* fetch raw signals, process and transform the signals. Exceptions and timeout control are

\* handled here.

\* @param query

\* @return Future[Option[Seq[Signal]]]

\*/

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

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

Stats

.trackItems(clientStatsReceiver) {

val rawSignals = getRawSignals(query.userId)

val signals = process(query, rawSignals)

signals

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

case e =>

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

EmptyResponse

}

}

/\*\*

\* Override this function to define how to fetch the raw signals from any store

\* Note that the RawSignalType is an intermediate data type intended to be small to avoid

\* big data chunks being passed over functions or being memcached.

\* @param userId

\* @return Future[Option[Seq[RawSignalType]]]

\*/

def getRawSignals(userId: UserId): Future[Option[Seq[RawSignalType]]]

/\*\*

\* Override this function to define how to process the raw signals and transform them to signals.

\* @param query

\* @param rawSignals

\* @return Future[Option[Seq[Signal]]]

\*/

def process(

query: Query,

rawSignals: Future[Option[Seq[RawSignalType]]]

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

}

object BaseSignalFetcher {

val Timeout: Duration = 20.milliseconds

val EmptyResponse: Option[Seq[Signal]] = Some(Seq.empty[Signal])

}