package com.twitter.tweetypie

package config

import com.twitter.servo.util.FutureArrow

import com.twitter.servo.util.RetryHandler

import com.twitter.servo.util.Scribe

import com.twitter.tweetypie.backends.LimiterService.Feature.MediaTagCreate

import com.twitter.tweetypie.backends.LimiterService.Feature.Updates

import com.twitter.tweetypie.client\_id.ClientIdHelper

import com.twitter.tweetypie.handler.TweetBuilder

import com.twitter.tweetypie.repository.TweetKeyFactory

import com.twitter.tweetypie.store.\_

import com.twitter.tweetypie.tflock.TFlockIndexer

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.tweetypie.util.RetryPolicyBuilder

import com.twitter.util.Timer

object TweetStores {

def apply(

settings: TweetServiceSettings,

statsReceiver: StatsReceiver,

timer: Timer,

deciderGates: TweetypieDeciderGates,

tweetKeyFactory: TweetKeyFactory,

clients: BackendClients,

caches: Caches,

asyncBuilder: ServiceInvocationBuilder,

hasMedia: Tweet => Boolean,

clientIdHelper: ClientIdHelper,

): TotalTweetStore = {

val deferredrpcRetryPolicy =

// retry all application exceptions for now. however, in the future, deferredrpc

// may throw a backpressure exception that should not be retried.

RetryPolicyBuilder.anyFailure(settings.deferredrpcBackoffs)

val asyncWriteRetryPolicy =

// currently retries all failures with the same back-off times. might need

// to update to handle backpressure exceptions differently.

RetryPolicyBuilder.anyFailure(settings.asyncWriteRetryBackoffs)

val replicatedEventRetryPolicy =

RetryPolicyBuilder.anyFailure(settings.replicatedEventCacheBackoffs)

val logLensStore =

LogLensStore(

tweetCreationsLogger = Logger("com.twitter.tweetypie.store.TweetCreations"),

tweetDeletionsLogger = Logger("com.twitter.tweetypie.store.TweetDeletions"),

tweetUndeletionsLogger = Logger("com.twitter.tweetypie.store.TweetUndeletions"),

tweetUpdatesLogger = Logger("com.twitter.tweetypie.store.TweetUpdates"),

clientIdHelper = clientIdHelper,

)

val tweetStoreStats = statsReceiver.scope("tweet\_store")

val tweetStatsStore = TweetStatsStore(tweetStoreStats.scope("stats"))

val asyncRetryConfig =

new TweetStore.AsyncRetry(

asyncWriteRetryPolicy,

deferredrpcRetryPolicy,

timer,

clients.asyncRetryTweetService,

Scribe(FailedAsyncWrite, "tweetypie\_failed\_async\_writes")

)(\_, \_)

val manhattanStore = {

val scopedStats = tweetStoreStats.scope("base")

ManhattanTweetStore(clients.tweetStorageClient)

.tracked(scopedStats)

.asyncRetry(asyncRetryConfig(scopedStats, ManhattanTweetStore.Action))

}

val cachingTweetStore = {

val cacheStats = tweetStoreStats.scope("caching")

CachingTweetStore(

tweetKeyFactory = tweetKeyFactory,

tweetCache = caches.tweetCache,

stats = cacheStats

).tracked(cacheStats)

.asyncRetry(asyncRetryConfig(cacheStats, CachingTweetStore.Action))

.replicatedRetry(RetryHandler.failuresOnly(replicatedEventRetryPolicy, timer, cacheStats))

}

val indexingStore = {

val indexingStats = tweetStoreStats.scope("indexing")

TweetIndexingStore(

new TFlockIndexer(

tflock = clients.tflockWriteClient,

hasMedia = hasMedia,

backgroundIndexingPriority = settings.backgroundIndexingPriority,

stats = indexingStats

)

).tracked(indexingStats)

.asyncRetry(asyncRetryConfig(indexingStats, TweetIndexingStore.Action))

}

val timelineUpdatingStore = {

val tlsScope = tweetStoreStats.scope("timeline\_updating")

TlsTimelineUpdatingStore(

processEvent2 = clients.timelineService.processEvent2,

hasMedia = hasMedia,

stats = tlsScope

).tracked(tlsScope)

.asyncRetry(asyncRetryConfig(tlsScope, TlsTimelineUpdatingStore.Action))

}

val guanoServiceStore = {

val guanoStats = tweetStoreStats.scope("guano")

GuanoServiceStore(clients.guano, guanoStats)

.tracked(guanoStats)

.asyncRetry(asyncRetryConfig(guanoStats, GuanoServiceStore.Action))

}

val mediaServiceStore = {

val mediaStats = tweetStoreStats.scope("media")

MediaServiceStore(clients.mediaClient.deleteMedia, clients.mediaClient.undeleteMedia)

.tracked(mediaStats)

.asyncRetry(asyncRetryConfig(mediaStats, MediaServiceStore.Action))

}

val userCountsUpdatingStore = {

val userCountsStats = tweetStoreStats.scope("user\_counts")

GizmoduckUserCountsUpdatingStore(clients.gizmoduck.incrCount, hasMedia)

.tracked(userCountsStats)

.ignoreFailures

}

val tweetCountsUpdatingStore = {

val cacheScope = statsReceiver.scope("tweet\_counts\_cache")

val tweetCountsStats = tweetStoreStats.scope("tweet\_counts")

val memcacheCountsStore = {

val lockingCacheCountsStore =

CachedCountsStore.fromLockingCache(caches.tweetCountsCache)

new AggregatingCachedCountsStore(

lockingCacheCountsStore,

timer,

settings.aggregatedTweetCountsFlushInterval,

settings.maxAggregatedCountsSize,

cacheScope

)

}

TweetCountsCacheUpdatingStore(memcacheCountsStore)

.tracked(tweetCountsStats)

.ignoreFailures

}

val replicatingStore = {

val replicateStats = tweetStoreStats.scope("replicate\_out")

ReplicatingTweetStore(

clients.replicationClient

).tracked(replicateStats)

.retry(RetryHandler.failuresOnly(deferredrpcRetryPolicy, timer, replicateStats))

.asyncRetry(asyncRetryConfig(replicateStats, ReplicatingTweetStore.Action))

.enabledBy(Gate.const(settings.enableReplication))

}

val scribeMediaTagStore =

ScribeMediaTagStore()

.tracked(tweetStoreStats.scope("scribe\_media\_tag\_store"))

val limiterStore =

LimiterStore(

clients.limiterService.incrementByOne(Updates),

clients.limiterService.increment(MediaTagCreate)

).tracked(tweetStoreStats.scope("limiter\_store"))

val geoSearchRequestIDStore = {

val statsScope = tweetStoreStats.scope("geo\_search\_request\_id")

GeoSearchRequestIDStore(FutureArrow(clients.geoRelevance.reportConversion \_))

.tracked(statsScope)

.asyncRetry(asyncRetryConfig(statsScope, GeoSearchRequestIDStore.Action))

}

val userGeotagUpdateStore = {

val geotagScope = tweetStoreStats.scope("gizmoduck\_user\_geotag\_updating")

GizmoduckUserGeotagUpdateStore(

clients.gizmoduck.modifyAndGet,

geotagScope

).tracked(geotagScope)

.asyncRetry(asyncRetryConfig(geotagScope, GizmoduckUserGeotagUpdateStore.Action))

}

val fanoutServiceStore = {

val fanoutStats = tweetStoreStats.scope("fanout\_service\_delivery")

FanoutServiceStore(clients.fanoutServiceClient, fanoutStats)

.tracked(fanoutStats)

.asyncRetry(asyncRetryConfig(fanoutStats, FanoutServiceStore.Action))

}

/\*\*

\* A store that converts Tweetypie TweetEvents to EventBus TweetEvents and sends each event to

\* the underlying FutureEffect[eventbus.TweetEvent]

\*/

val eventBusEnqueueStore = {

val enqueueStats = tweetStoreStats.scope("event\_bus\_enqueueing")

val enqueueEffect = FutureEffect[TweetEvent](clients.tweetEventsPublisher.publish)

TweetEventBusStore(

enqueueEffect

).tracked(enqueueStats)

.asyncRetry(asyncRetryConfig(enqueueStats, AsyncWriteAction.EventBusEnqueue))

}

val retweetArchivalEnqueueStore = {

val enqueueStats = tweetStoreStats.scope("retweet\_archival\_enqueueing")

val enqueueEffect = FutureEffect(clients.retweetArchivalEventPublisher.publish)

RetweetArchivalEnqueueStore(enqueueEffect)

.tracked(enqueueStats)

.asyncRetry(asyncRetryConfig(enqueueStats, AsyncWriteAction.RetweetArchivalEnqueue))

}

val asyncEnqueueStore = {

val asyncEnqueueStats = tweetStoreStats.scope("async\_enqueueing")

AsyncEnqueueStore(

asyncBuilder.asyncVia(clients.asyncTweetService).service,

TweetBuilder.scrubUserInAsyncInserts,

TweetBuilder.scrubSourceTweetInAsyncInserts,

TweetBuilder.scrubSourceUserInAsyncInserts

).tracked(asyncEnqueueStats)

.retry(RetryHandler.failuresOnly(deferredrpcRetryPolicy, timer, asyncEnqueueStats))

}

val insertTweetStore =

InsertTweet.Store(

logLensStore = logLensStore,

manhattanStore = manhattanStore,

tweetStatsStore = tweetStatsStore,

cachingTweetStore = cachingTweetStore,

limiterStore = limiterStore,

asyncEnqueueStore = asyncEnqueueStore,

userCountsUpdatingStore = userCountsUpdatingStore,

tweetCountsUpdatingStore = tweetCountsUpdatingStore

)

val asyncInsertStore =

AsyncInsertTweet.Store(

replicatingStore = replicatingStore,

indexingStore = indexingStore,

tweetCountsUpdatingStore = tweetCountsUpdatingStore,

timelineUpdatingStore = timelineUpdatingStore,

eventBusEnqueueStore = eventBusEnqueueStore,

fanoutServiceStore = fanoutServiceStore,

scribeMediaTagStore = scribeMediaTagStore,

userGeotagUpdateStore = userGeotagUpdateStore,

geoSearchRequestIDStore = geoSearchRequestIDStore

)

val replicatedInsertTweetStore =

ReplicatedInsertTweet.Store(

cachingTweetStore = cachingTweetStore,

tweetCountsUpdatingStore = tweetCountsUpdatingStore

)

val deleteTweetStore =

DeleteTweet.Store(

cachingTweetStore = cachingTweetStore,

asyncEnqueueStore = asyncEnqueueStore,

userCountsUpdatingStore = userCountsUpdatingStore,

tweetCountsUpdatingStore = tweetCountsUpdatingStore,

logLensStore = logLensStore

)

val asyncDeleteTweetStore =

AsyncDeleteTweet.Store(

manhattanStore = manhattanStore,

cachingTweetStore = cachingTweetStore,

replicatingStore = replicatingStore,

indexingStore = indexingStore,

eventBusEnqueueStore = eventBusEnqueueStore,

timelineUpdatingStore = timelineUpdatingStore,

tweetCountsUpdatingStore = tweetCountsUpdatingStore,

guanoServiceStore = guanoServiceStore,

mediaServiceStore = mediaServiceStore

)

val replicatedDeleteTweetStore =

ReplicatedDeleteTweet.Store(

cachingTweetStore = cachingTweetStore,

tweetCountsUpdatingStore = tweetCountsUpdatingStore

)

val incrBookmarkCountStore =

IncrBookmarkCount.Store(

asyncEnqueueStore = asyncEnqueueStore,

replicatingStore = replicatingStore

)

val asyncIncrBookmarkCountStore =

AsyncIncrBookmarkCount.Store(

tweetCountsUpdatingStore = tweetCountsUpdatingStore

)

val replicatedIncrBookmarkCountStore =

ReplicatedIncrBookmarkCount.Store(

tweetCountsUpdatingStore = tweetCountsUpdatingStore

)

val incrFavCountStore =

IncrFavCount.Store(

asyncEnqueueStore = asyncEnqueueStore,

replicatingStore = replicatingStore

)

val asyncIncrFavCountStore =

AsyncIncrFavCount.Store(

tweetCountsUpdatingStore = tweetCountsUpdatingStore

)

val replicatedIncrFavCountStore =

ReplicatedIncrFavCount.Store(

tweetCountsUpdatingStore = tweetCountsUpdatingStore

)

val scrubGeoStore =

ScrubGeo.Store(

logLensStore = logLensStore,

manhattanStore = manhattanStore,

cachingTweetStore = cachingTweetStore,

eventBusEnqueueStore = eventBusEnqueueStore,

replicatingStore = replicatingStore

)

val replicatedScrubGeoStore =

ReplicatedScrubGeo.Store(

cachingTweetStore = cachingTweetStore

)

val takedownStore =

Takedown.Store(

logLensStore = logLensStore,

manhattanStore = manhattanStore,

cachingTweetStore = cachingTweetStore,

asyncEnqueueStore = asyncEnqueueStore

)

val asyncTakedownStore =

AsyncTakedown.Store(

replicatingStore = replicatingStore,

guanoStore = guanoServiceStore,

eventBusEnqueueStore = eventBusEnqueueStore

)

val replicatedTakedownStore =

ReplicatedTakedown.Store(

cachingTweetStore = cachingTweetStore

)

val updatePossiblySensitiveTweetStore =

UpdatePossiblySensitiveTweet.Store(

manhattanStore = manhattanStore,

cachingTweetStore = cachingTweetStore,

logLensStore = logLensStore,

asyncEnqueueStore = asyncEnqueueStore

)

val asyncUpdatePossiblySensitiveTweetStore =

AsyncUpdatePossiblySensitiveTweet.Store(

manhattanStore = manhattanStore,

cachingTweetStore = cachingTweetStore,

replicatingStore = replicatingStore,

guanoStore = guanoServiceStore,

eventBusStore = eventBusEnqueueStore

)

val replicatedUpdatePossiblySensitiveTweetStore =

ReplicatedUpdatePossiblySensitiveTweet.Store(

cachingTweetStore = cachingTweetStore

)

val setAdditionalFieldsStore =

SetAdditionalFields.Store(

manhattanStore = manhattanStore,

cachingTweetStore = cachingTweetStore,

asyncEnqueueStore = asyncEnqueueStore,

logLensStore = logLensStore

)

val asyncSetAdditionalFieldsStore =

AsyncSetAdditionalFields.Store(

replicatingStore = replicatingStore,

eventBusEnqueueStore = eventBusEnqueueStore

)

val replicatedSetAdditionalFieldsStore =

ReplicatedSetAdditionalFields.Store(

cachingTweetStore = cachingTweetStore

)

val setRetweetVisibilityStore =

SetRetweetVisibility.Store(asyncEnqueueStore = asyncEnqueueStore)

val asyncSetRetweetVisibilityStore =

AsyncSetRetweetVisibility.Store(

tweetIndexingStore = indexingStore,

tweetCountsCacheUpdatingStore = tweetCountsUpdatingStore,

replicatingTweetStore = replicatingStore,

retweetArchivalEnqueueStore = retweetArchivalEnqueueStore

)

val replicatedSetRetweetVisibilityStore =

ReplicatedSetRetweetVisibility.Store(

tweetCountsCacheUpdatingStore = tweetCountsUpdatingStore

)

val deleteAdditionalFieldsStore =

DeleteAdditionalFields.Store(

cachingTweetStore = cachingTweetStore,

asyncEnqueueStore = asyncEnqueueStore,

logLensStore = logLensStore

)

val asyncDeleteAdditionalFieldsStore =

AsyncDeleteAdditionalFields.Store(

manhattanStore = manhattanStore,

cachingTweetStore = cachingTweetStore,

replicatingStore = replicatingStore,

eventBusEnqueueStore = eventBusEnqueueStore

)

val replicatedDeleteAdditionalFieldsStore =

ReplicatedDeleteAdditionalFields.Store(

cachingTweetStore = cachingTweetStore

)

/\*

\* This composed store handles all synchronous side effects of an undelete

\* but does not execute the undeletion.

\*

\* This store is executed after the actual undelete request succeeds.

\* The undeletion request is initiated by Undelete.apply()

\*/

val undeleteTweetStore =

UndeleteTweet.Store(

logLensStore = logLensStore,

cachingTweetStore = cachingTweetStore,

tweetCountsUpdatingStore = tweetCountsUpdatingStore,

asyncEnqueueStore = asyncEnqueueStore

)

val asyncUndeleteTweetStore =

AsyncUndeleteTweet.Store(

cachingTweetStore = cachingTweetStore,

eventBusEnqueueStore = eventBusEnqueueStore,

indexingStore = indexingStore,

replicatingStore = replicatingStore,

mediaServiceStore = mediaServiceStore,

timelineUpdatingStore = timelineUpdatingStore

)

val replicatedUndeleteTweetStore =

ReplicatedUndeleteTweet.Store(

cachingTweetStore = cachingTweetStore,

tweetCountsUpdatingStore = tweetCountsUpdatingStore

)

val flushStore =

Flush.Store(

cachingTweetStore = cachingTweetStore,

tweetCountsUpdatingStore = tweetCountsUpdatingStore

)

val scrubGeoUpdateUserTimestampStore =

ScrubGeoUpdateUserTimestamp.Store(

cache = caches.geoScrubCache,

setInManhattan = clients.geoScrubEventStore.setGeoScrubTimestamp,

geotagUpdateStore = userGeotagUpdateStore,

tweetEventBusStore = eventBusEnqueueStore

)

val quotedTweetDeleteStore =

QuotedTweetDelete.Store(

eventBusEnqueueStore = eventBusEnqueueStore

)

val quotedTweetTakedownStore =

QuotedTweetTakedown.Store(

eventBusEnqueueStore = eventBusEnqueueStore

)

new TotalTweetStore {

val asyncDeleteAdditionalFields: FutureEffect[AsyncDeleteAdditionalFields.Event] =

asyncDeleteAdditionalFieldsStore.asyncDeleteAdditionalFields

val asyncDeleteTweet: FutureEffect[AsyncDeleteTweet.Event] =

asyncDeleteTweetStore.asyncDeleteTweet

val asyncIncrBookmarkCount: FutureEffect[AsyncIncrBookmarkCount.Event] =

asyncIncrBookmarkCountStore.asyncIncrBookmarkCount

val asyncIncrFavCount: FutureEffect[AsyncIncrFavCount.Event] =

asyncIncrFavCountStore.asyncIncrFavCount

val asyncInsertTweet: FutureEffect[AsyncInsertTweet.Event] = asyncInsertStore.asyncInsertTweet

val asyncSetAdditionalFields: FutureEffect[AsyncSetAdditionalFields.Event] =

asyncSetAdditionalFieldsStore.asyncSetAdditionalFields

val asyncSetRetweetVisibility: FutureEffect[AsyncSetRetweetVisibility.Event] =

asyncSetRetweetVisibilityStore.asyncSetRetweetVisibility

val asyncTakedown: FutureEffect[AsyncTakedown.Event] = asyncTakedownStore.asyncTakedown

val asyncUndeleteTweet: FutureEffect[AsyncUndeleteTweet.Event] =

asyncUndeleteTweetStore.asyncUndeleteTweet

val asyncUpdatePossiblySensitiveTweet: FutureEffect[AsyncUpdatePossiblySensitiveTweet.Event] =

asyncUpdatePossiblySensitiveTweetStore.asyncUpdatePossiblySensitiveTweet

val deleteAdditionalFields: FutureEffect[DeleteAdditionalFields.Event] =

deleteAdditionalFieldsStore.deleteAdditionalFields

val deleteTweet: FutureEffect[DeleteTweet.Event] = deleteTweetStore.deleteTweet

val flush: FutureEffect[Flush.Event] = flushStore.flush

val incrBookmarkCount: FutureEffect[IncrBookmarkCount.Event] =

incrBookmarkCountStore.incrBookmarkCount

val incrFavCount: FutureEffect[IncrFavCount.Event] = incrFavCountStore.incrFavCount

val insertTweet: FutureEffect[InsertTweet.Event] = insertTweetStore.insertTweet

val quotedTweetDelete: FutureEffect[QuotedTweetDelete.Event] =

quotedTweetDeleteStore.quotedTweetDelete

val quotedTweetTakedown: FutureEffect[QuotedTweetTakedown.Event] =

quotedTweetTakedownStore.quotedTweetTakedown

val replicatedDeleteAdditionalFields: FutureEffect[ReplicatedDeleteAdditionalFields.Event] =

replicatedDeleteAdditionalFieldsStore.replicatedDeleteAdditionalFields

val replicatedDeleteTweet: FutureEffect[ReplicatedDeleteTweet.Event] =

replicatedDeleteTweetStore.replicatedDeleteTweet

val replicatedIncrBookmarkCount: FutureEffect[ReplicatedIncrBookmarkCount.Event] =

replicatedIncrBookmarkCountStore.replicatedIncrBookmarkCount

val replicatedIncrFavCount: FutureEffect[ReplicatedIncrFavCount.Event] =

replicatedIncrFavCountStore.replicatedIncrFavCount

val replicatedInsertTweet: FutureEffect[ReplicatedInsertTweet.Event] =

replicatedInsertTweetStore.replicatedInsertTweet

val replicatedScrubGeo: FutureEffect[ReplicatedScrubGeo.Event] =

replicatedScrubGeoStore.replicatedScrubGeo

val replicatedSetAdditionalFields: FutureEffect[ReplicatedSetAdditionalFields.Event] =

replicatedSetAdditionalFieldsStore.replicatedSetAdditionalFields

val replicatedSetRetweetVisibility: FutureEffect[ReplicatedSetRetweetVisibility.Event] =

replicatedSetRetweetVisibilityStore.replicatedSetRetweetVisibility

val replicatedTakedown: FutureEffect[ReplicatedTakedown.Event] =

replicatedTakedownStore.replicatedTakedown

val replicatedUndeleteTweet: FutureEffect[ReplicatedUndeleteTweet.Event] =

replicatedUndeleteTweetStore.replicatedUndeleteTweet

val replicatedUpdatePossiblySensitiveTweet: FutureEffect[

ReplicatedUpdatePossiblySensitiveTweet.Event

] =

replicatedUpdatePossiblySensitiveTweetStore.replicatedUpdatePossiblySensitiveTweet

val retryAsyncDeleteAdditionalFields: FutureEffect[

TweetStoreRetryEvent[AsyncDeleteAdditionalFields.Event]

] =

asyncDeleteAdditionalFieldsStore.retryAsyncDeleteAdditionalFields

val retryAsyncDeleteTweet: FutureEffect[TweetStoreRetryEvent[AsyncDeleteTweet.Event]] =

asyncDeleteTweetStore.retryAsyncDeleteTweet

val retryAsyncInsertTweet: FutureEffect[TweetStoreRetryEvent[AsyncInsertTweet.Event]] =

asyncInsertStore.retryAsyncInsertTweet

val retryAsyncSetAdditionalFields: FutureEffect[

TweetStoreRetryEvent[AsyncSetAdditionalFields.Event]

] =

asyncSetAdditionalFieldsStore.retryAsyncSetAdditionalFields

val retryAsyncSetRetweetVisibility: FutureEffect[

TweetStoreRetryEvent[AsyncSetRetweetVisibility.Event]

] =

asyncSetRetweetVisibilityStore.retryAsyncSetRetweetVisibility

val retryAsyncTakedown: FutureEffect[TweetStoreRetryEvent[AsyncTakedown.Event]] =

asyncTakedownStore.retryAsyncTakedown

val retryAsyncUndeleteTweet: FutureEffect[TweetStoreRetryEvent[AsyncUndeleteTweet.Event]] =

asyncUndeleteTweetStore.retryAsyncUndeleteTweet

val retryAsyncUpdatePossiblySensitiveTweet: FutureEffect[

TweetStoreRetryEvent[AsyncUpdatePossiblySensitiveTweet.Event]

] =

asyncUpdatePossiblySensitiveTweetStore.retryAsyncUpdatePossiblySensitiveTweet

val scrubGeo: FutureEffect[ScrubGeo.Event] = scrubGeoStore.scrubGeo

val setAdditionalFields: FutureEffect[SetAdditionalFields.Event] =

setAdditionalFieldsStore.setAdditionalFields

val setRetweetVisibility: FutureEffect[SetRetweetVisibility.Event] =

setRetweetVisibilityStore.setRetweetVisibility

val takedown: FutureEffect[Takedown.Event] = takedownStore.takedown

val undeleteTweet: FutureEffect[UndeleteTweet.Event] = undeleteTweetStore.undeleteTweet

val updatePossiblySensitiveTweet: FutureEffect[UpdatePossiblySensitiveTweet.Event] =

updatePossiblySensitiveTweetStore.updatePossiblySensitiveTweet

val scrubGeoUpdateUserTimestamp: FutureEffect[ScrubGeoUpdateUserTimestamp.Event] =

scrubGeoUpdateUserTimestampStore.scrubGeoUpdateUserTimestamp

}

}

}