package com.twitter.tweetypie

package config

import com.twitter.coreservices.failed\_task.writer.FailedTaskWriter

import com.twitter.featureswitches.v2.FeatureSwitches

import com.twitter.flockdb.client.\_

import com.twitter.servo.forked

import com.twitter.servo.util.FutureArrow

import com.twitter.servo.util.Scribe

import com.twitter.stitch.Stitch

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

import com.twitter.tweetypie.handler.\_

import com.twitter.tweetypie.repository.\_

import com.twitter.tweetypie.service.ReplicatingTweetService

import com.twitter.tweetypie.service.\_

import com.twitter.tweetypie.storage.TweetStorageClient

import com.twitter.tweetypie.storage.TweetStorageClient.GetTweet

import com.twitter.tweetypie.store.\_

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.util.Activity

import com.twitter.util.Timer

/\*\*

\* Builds a fully configured ThriftTweetService instance.

\*

\* The core of the tweet service is a DispatchingTweetService, which is responsible

\* for dispatching requests to underlying handlers and stores.

\* The DispatchingTweetService instance is wrapped in:

\* - ObservedTweetService (adds stats counting)

\* - ClientHandlingTweetService (authentication, exception handling, etc)

\* - ReplicatingTweetService (replicates some reads)

\*

\* TweetServiceBuilder returns an Activity[ThriftTweetService] which updates

\* on config changes. See DynamicConfig.scala for more details.

\*/

object TweetServiceBuilder {

def apply(

settings: TweetServiceSettings,

statsReceiver: StatsReceiver,

timer: Timer,

deciderGates: TweetypieDeciderGates,

featureSwitchesWithExperiments: FeatureSwitches,

featureSwitchesWithoutExperiments: FeatureSwitches,

backendClients: BackendClients,

clientIdHelper: ClientIdHelper,

): Activity[ThriftTweetService] = {

// a forward reference, will be set to the DispatchingTweetService once created

val syncTweetService = new MutableTweetServiceProxy(null)

val tweetServiceScope = statsReceiver.scope("tweet\_service")

val dispatchingTweetService =

DispatchingTweetServiceBuilder(

settings,

statsReceiver,

tweetServiceScope,

syncTweetService,

timer,

deciderGates,

featureSwitchesWithExperiments,

featureSwitchesWithoutExperiments,

backendClients,

clientIdHelper,

)

val failureLoggingTweetService =

// Add the failure writing inside of the authorization filter so

// that we don't write out the failures when authorization fails.

new FailureLoggingTweetService(

failedTaskWriter = FailedTaskWriter("tweetypie\_service\_failures", identity),

underlying = dispatchingTweetService

)

val observedTweetService =

new ObservedTweetService(failureLoggingTweetService, tweetServiceScope, clientIdHelper)

// Every time config is updated, create a new tweet service. Only

// ClientHandlingTweetService and ReplicatingTweetService need to

// be recreated, as the underlying TweetServices above don't depend

// on the config.

DynamicConfig(

statsReceiver.scope("dynamic\_config"),

backendClients.configBus,

settings

).map { dynamicConfig =>

val clientHandlingTweetService =

new ClientHandlingTweetService(

observedTweetService,

tweetServiceScope,

dynamicConfig.loadShedEligible,

deciderGates.shedReadTrafficVoluntarily,

ClientHandlingTweetServiceAuthorizer(

settings = settings,

dynamicConfig = dynamicConfig,

statsReceiver = statsReceiver

),

GetTweetsAuthorizer(

config = dynamicConfig,

maxRequestSize = settings.maxGetTweetsRequestSize,

instanceCount = settings.instanceCount,

enforceRateLimitedClients = deciderGates.enforceRateLimitedClients,

maxRequestWidthEnabled = deciderGates.maxRequestWidthEnabled,

statsReceiver = tweetServiceScope.scope("get\_tweets"),

),

GetTweetFieldsAuthorizer(

config = dynamicConfig,

maxRequestSize = settings.maxGetTweetsRequestSize,

instanceCount = settings.instanceCount,

enforceRateLimitedClients = deciderGates.enforceRateLimitedClients,

maxRequestWidthEnabled = deciderGates.maxRequestWidthEnabled,

statsReceiver = tweetServiceScope.scope("get\_tweet\_fields"),

),

RequestSizeAuthorizer(settings.maxRequestSize, deciderGates.maxRequestWidthEnabled),

clientIdHelper,

)

syncTweetService.underlying = clientHandlingTweetService

val replicatingService =

if (!settings.enableReplication)

clientHandlingTweetService

else {

new ReplicatingTweetService(

underlying = clientHandlingTweetService,

replicationTargets = backendClients.lowQoSReplicationClients,

executor = new forked.QueueExecutor(

100,

statsReceiver.scope("replicating\_tweet\_service")

),

)

}

replicatingService

}

}

}

object DispatchingTweetServiceBuilder {

val hasMedia: Tweet => Boolean = MediaIndexHelper(Resources.loadPartnerMediaRegexes())

def apply(

settings: TweetServiceSettings,

statsReceiver: StatsReceiver,

tweetServiceScope: StatsReceiver,

syncTweetService: ThriftTweetService,

timer: Timer,

deciderGates: TweetypieDeciderGates,

featureSwitchesWithExperiments: FeatureSwitches,

featureSwitchesWithoutExperiments: FeatureSwitches,

backendClients: BackendClients,

clientIdHelper: ClientIdHelper,

): ThriftTweetService = {

val (syncInvocationBuilder, asyncInvocationBuilder) = {

val b =

new ServiceInvocationBuilder(syncTweetService, settings.simulateDeferredrpcCallbacks)

(b.withClientId(settings.thriftClientId), b.withClientId(settings.deferredrpcClientId))

}

val tweetKeyFactory = TweetKeyFactory(settings.tweetKeyCacheVersion)

val caches =

if (!settings.withCache)

Caches.NoCache

else

Caches(

settings = settings,

stats = statsReceiver,

timer = timer,

clients = backendClients,

tweetKeyFactory = tweetKeyFactory,

deciderGates = deciderGates,

clientIdHelper = clientIdHelper,

)

val logicalRepos =

LogicalRepositories(

settings = settings,

stats = statsReceiver,

timer = timer,

deciderGates = deciderGates,

external = new ExternalServiceRepositories(

clients = backendClients,

statsReceiver = statsReceiver,

settings = settings,

clientIdHelper = clientIdHelper,

),

caches = caches,

stratoClient = backendClients.stratoserverClient,

hasMedia = hasMedia,

clientIdHelper = clientIdHelper,

featureSwitchesWithoutExperiments = featureSwitchesWithoutExperiments,

)

val tweetCreationLock =

new CacheBasedTweetCreationLock(

cache = caches.tweetCreateLockerCache,

maxTries = 3,

stats = statsReceiver.scope("tweet\_save").scope("locker"),

logUniquenessId =

if (settings.scribeUniquenessIds) CacheBasedTweetCreationLock.ScribeUniquenessId

else CacheBasedTweetCreationLock.LogUniquenessId

)

val tweetStores =

TweetStores(

settings = settings,

statsReceiver = statsReceiver,

timer = timer,

deciderGates = deciderGates,

tweetKeyFactory = tweetKeyFactory,

clients = backendClients,

caches = caches,

asyncBuilder = asyncInvocationBuilder,

hasMedia = hasMedia,

clientIdHelper = clientIdHelper,

)

val tweetDeletePathHandler =

new DefaultTweetDeletePathHandler(

tweetServiceScope,

logicalRepos.tweetResultRepo,

logicalRepos.optionalUserRepo,

logicalRepos.stratoSafetyLabelsRepo,

logicalRepos.lastQuoteOfQuoterRepo,

tweetStores,

getPerspectives = backendClients.timelineService.getPerspectives,

)

val tweetBuilders =

TweetBuilders(

settings = settings,

statsReceiver = statsReceiver,

deciderGates = deciderGates,

featureSwitchesWithExperiments = featureSwitchesWithExperiments,

clients = backendClients,

caches = caches,

repos = logicalRepos,

tweetStore = tweetStores,

hasMedia = hasMedia,

unretweetEdits = tweetDeletePathHandler.unretweetEdits,

)

val hydrateTweetForInsert =

WritePathHydration.hydrateTweet(

logicalRepos.tweetHydrators.hydrator,

statsReceiver.scope("insert\_tweet")

)

val defaultTweetQueryOptions = TweetQuery.Options(include = GetTweetsHandler.BaseInclude)

val parentUserIdRepo: ParentUserIdRepository.Type =

ParentUserIdRepository(

tweetRepo = logicalRepos.tweetRepo

)

val undeleteTweetHandler =

UndeleteTweetHandlerBuilder(

backendClients.tweetStorageClient,

logicalRepos,

tweetStores,

parentUserIdRepo,

statsReceiver

)

val eraseUserTweetsHandler =

EraseUserTweetsHandlerBuilder(

backendClients,

asyncInvocationBuilder,

deciderGates,

settings,

timer,

tweetDeletePathHandler,

tweetServiceScope

)

val setRetweetVisibilityHandler =

SetRetweetVisibilityHandler(

tweetGetter =

TweetRepository.tweetGetter(logicalRepos.optionalTweetRepo, defaultTweetQueryOptions),

tweetStores.setRetweetVisibility

)

val takedownHandler =

TakedownHandlerBuilder(

logicalRepos = logicalRepos,

tweetStores = tweetStores

)

val updatePossiblySensitiveTweetHandler =

UpdatePossiblySensitiveTweetHandler(

HandlerError.getRequired(

TweetRepository.tweetGetter(logicalRepos.optionalTweetRepo, defaultTweetQueryOptions),

HandlerError.tweetNotFoundException

),

HandlerError.getRequired(

FutureArrow(

UserRepository

.userGetter(

logicalRepos.optionalUserRepo,

UserQueryOptions(Set(UserField.Safety), UserVisibility.All)

)

.compose(UserKey.byId)

),

HandlerError.userNotFoundException

),

tweetStores.updatePossiblySensitiveTweet

)

val userTakedownHandler =

UserTakedownHandlerBuilder(

logicalRepos = logicalRepos,

tweetStores = tweetStores,

stats = tweetServiceScope

)

val getDeletedTweetsHandler =

GetDeletedTweetsHandler(

getDeletedTweets = backendClients.tweetStorageClient.getDeletedTweets,

tweetsExist =

GetDeletedTweetsHandler.tweetsExist(backendClients.tweetStorageClient.getTweet),

stats = tweetServiceScope.scope("get\_deleted\_tweets\_handler")

)

val hydrateQuotedTweet =

WritePathHydration.hydrateQuotedTweet(

logicalRepos.optionalTweetRepo,

logicalRepos.optionalUserRepo,

logicalRepos.quoterHasAlreadyQuotedRepo

)

val deleteLocationDataHandler =

DeleteLocationDataHandler(

backendClients.geoScrubEventStore.getGeoScrubTimestamp,

Scribe(DeleteLocationData, "tweetypie\_delete\_location\_data"),

backendClients.deleteLocationDataPublisher

)

val getStoredTweetsHandler = GetStoredTweetsHandler(logicalRepos.tweetResultRepo)

val getStoredTweetsByUserHandler = GetStoredTweetsByUserHandler(

getStoredTweetsHandler = getStoredTweetsHandler,

getStoredTweet = backendClients.tweetStorageClient.getStoredTweet,

selectPage = FutureArrow { select =>

backendClients.tflockReadClient

.selectPage(select, Some(settings.getStoredTweetsByUserPageSize))

},

maxPages = settings.getStoredTweetsByUserMaxPages

)

val getTweetsHandler =

GetTweetsHandler(

logicalRepos.tweetResultRepo,

logicalRepos.containerAsGetTweetResultRepo,

logicalRepos.deletedTweetVisibilityRepo,

statsReceiver.scope("read\_path"),

deciderGates.shouldMaterializeContainers

)

val getTweetFieldsHandler =

GetTweetFieldsHandler(

logicalRepos.tweetResultRepo,

logicalRepos.deletedTweetVisibilityRepo,

logicalRepos.containerAsGetTweetFieldsResultRepo,

statsReceiver.scope("read\_path"),

deciderGates.shouldMaterializeContainers

)

val unretweetHandler =

UnretweetHandler(

tweetDeletePathHandler.deleteTweets,

backendClients.timelineService.getPerspectives,

tweetDeletePathHandler.unretweetEdits,

logicalRepos.tweetRepo,

)

val hydrateInsertEvent =

WritePathHydration.hydrateInsertTweetEvent(

hydrateTweet = hydrateTweetForInsert,

hydrateQuotedTweet = hydrateQuotedTweet

)

val scrubGeoUpdateUserTimestampBuilder =

ScrubGeoEventBuilder.UpdateUserTimestamp(

stats = tweetServiceScope.scope("scrub\_geo\_update\_user\_timestamp"),

userRepo = logicalRepos.optionalUserRepo

)

val scrubGeoScrubTweetsBuilder =

ScrubGeoEventBuilder.ScrubTweets(

stats = tweetServiceScope.scope("scrub\_geo"),

userRepo = logicalRepos.optionalUserRepo

)

val handlerFilter =

PostTweet

.DuplicateHandler(

tweetCreationLock = tweetCreationLock,

getTweets = getTweetsHandler,

stats = statsReceiver.scope("duplicate")

)

.andThen(PostTweet.RescueTweetCreateFailure)

.andThen(PostTweet.LogFailures)

val postTweetHandler =

handlerFilter[PostTweetRequest](

PostTweet.Handler(

tweetBuilder = tweetBuilders.tweetBuilder,

hydrateInsertEvent = hydrateInsertEvent,

tweetStore = tweetStores,

)

)

val postRetweetHandler =

handlerFilter[RetweetRequest](

PostTweet.Handler(

tweetBuilder = tweetBuilders.retweetBuilder,

hydrateInsertEvent = hydrateInsertEvent,

tweetStore = tweetStores,

)

)

val quotedTweetDeleteBuilder: QuotedTweetDeleteEventBuilder.Type =

QuotedTweetDeleteEventBuilder(logicalRepos.optionalTweetRepo)

val quotedTweetTakedownBuilder: QuotedTweetTakedownEventBuilder.Type =

QuotedTweetTakedownEventBuilder(logicalRepos.optionalTweetRepo)

val setAdditionalFieldsBuilder: SetAdditionalFieldsBuilder.Type =

SetAdditionalFieldsBuilder(

tweetRepo = logicalRepos.tweetRepo

)

val asyncSetAdditionalFieldsBuilder: AsyncSetAdditionalFieldsBuilder.Type =

AsyncSetAdditionalFieldsBuilder(

userRepo = logicalRepos.userRepo

)

val deleteAdditionalFieldsBuilder: DeleteAdditionalFieldsBuilder.Type =

DeleteAdditionalFieldsBuilder(

tweetRepo = logicalRepos.tweetRepo

)

val asyncDeleteAdditionalFieldsBuilder: AsyncDeleteAdditionalFieldsBuilder.Type =

AsyncDeleteAdditionalFieldsBuilder(

userRepo = logicalRepos.userRepo

)

new DispatchingTweetService(

asyncDeleteAdditionalFieldsBuilder = asyncDeleteAdditionalFieldsBuilder,

asyncSetAdditionalFieldsBuilder = asyncSetAdditionalFieldsBuilder,

deleteAdditionalFieldsBuilder = deleteAdditionalFieldsBuilder,

deleteLocationDataHandler = deleteLocationDataHandler,

deletePathHandler = tweetDeletePathHandler,

eraseUserTweetsHandler = eraseUserTweetsHandler,

getDeletedTweetsHandler = getDeletedTweetsHandler,

getStoredTweetsHandler = getStoredTweetsHandler,

getStoredTweetsByUserHandler = getStoredTweetsByUserHandler,

getTweetsHandler = getTweetsHandler,

getTweetFieldsHandler = getTweetFieldsHandler,

getTweetCountsHandler = GetTweetCountsHandler(logicalRepos.tweetCountsRepo),

postTweetHandler = postTweetHandler,

postRetweetHandler = postRetweetHandler,

quotedTweetDeleteBuilder = quotedTweetDeleteBuilder,

quotedTweetTakedownBuilder = quotedTweetTakedownBuilder,

scrubGeoUpdateUserTimestampBuilder = scrubGeoUpdateUserTimestampBuilder,

scrubGeoScrubTweetsBuilder = scrubGeoScrubTweetsBuilder,

setAdditionalFieldsBuilder = setAdditionalFieldsBuilder,

setRetweetVisibilityHandler = setRetweetVisibilityHandler,

statsReceiver = statsReceiver,

takedownHandler = takedownHandler,

tweetStore = tweetStores,

undeleteTweetHandler = undeleteTweetHandler,

unretweetHandler = unretweetHandler,

updatePossiblySensitiveTweetHandler = updatePossiblySensitiveTweetHandler,

userTakedownHandler = userTakedownHandler,

clientIdHelper = clientIdHelper,

)

}

}

object TakedownHandlerBuilder {

type Type = FutureArrow[TakedownRequest, Unit]

def apply(logicalRepos: LogicalRepositories, tweetStores: TotalTweetStore) =

TakedownHandler(

getTweet = HandlerError.getRequired(

tweetGetter(logicalRepos),

HandlerError.tweetNotFoundException

),

getUser = HandlerError.getRequired(

userGetter(logicalRepos),

HandlerError.userNotFoundException

),

writeTakedown = tweetStores.takedown

)

def tweetGetter(logicalRepos: LogicalRepositories): FutureArrow[TweetId, Option[Tweet]] =

FutureArrow(

TweetRepository.tweetGetter(

logicalRepos.optionalTweetRepo,

TweetQuery.Options(

include = GetTweetsHandler.BaseInclude.also(

tweetFields = Set(

Tweet.TweetypieOnlyTakedownCountryCodesField.id,

Tweet.TweetypieOnlyTakedownReasonsField.id

)

)

)

)

)

def userGetter(logicalRepos: LogicalRepositories): FutureArrow[UserId, Option[User]] =

FutureArrow(

UserRepository

.userGetter(

logicalRepos.optionalUserRepo,

UserQueryOptions(

Set(UserField.Roles, UserField.Safety, UserField.Takedowns),

UserVisibility.All

)

)

.compose(UserKey.byId)

)

}

object UserTakedownHandlerBuilder {

def apply(

logicalRepos: LogicalRepositories,

tweetStores: TotalTweetStore,

stats: StatsReceiver

): UserTakedownHandler.Type =

UserTakedownHandler(

getTweet = TakedownHandlerBuilder.tweetGetter(logicalRepos),

tweetTakedown = tweetStores.takedown,

)

}

object EraseUserTweetsHandlerBuilder {

def apply(

backendClients: BackendClients,

asyncInvocationBuilder: ServiceInvocationBuilder,

deciderGates: TweetypieDeciderGates,

settings: TweetServiceSettings,

timer: Timer,

tweetDeletePathHandler: DefaultTweetDeletePathHandler,

tweetServiceScope: StatsReceiver

): EraseUserTweetsHandler =

EraseUserTweetsHandler(

selectPage(backendClients, settings),

deleteTweet(tweetDeletePathHandler),

eraseUserTweets(backendClients, asyncInvocationBuilder),

tweetServiceScope.scope("erase\_user\_tweets"),

sleep(deciderGates, settings, timer)

)

def selectPage(

backendClients: BackendClients,

settings: TweetServiceSettings

): FutureArrow[Select[StatusGraph], PageResult[Long]] =

FutureArrow(

backendClients.tflockWriteClient.selectPage(\_, Some(settings.eraseUserTweetsPageSize))

)

def deleteTweet(

tweetDeletePathHandler: DefaultTweetDeletePathHandler

): FutureEffect[(TweetId, UserId)] =

FutureEffect[(TweetId, UserId)] {

case (tweetId, expectedUserId) =>

tweetDeletePathHandler

.internalDeleteTweets(

request = DeleteTweetsRequest(

Seq(tweetId),

isUserErasure = true,

expectedUserId = Some(expectedUserId)

),

byUserId = None,

authenticatedUserId = None,

validate = tweetDeletePathHandler.validateTweetsForUserErasureDaemon

)

.unit

}

def eraseUserTweets(

backendClients: BackendClients,

asyncInvocationBuilder: ServiceInvocationBuilder

): FutureArrow[AsyncEraseUserTweetsRequest, Unit] =

asyncInvocationBuilder

.asyncVia(backendClients.asyncTweetDeletionService)

.method(\_.asyncEraseUserTweets)

def sleep(

deciderGates: TweetypieDeciderGates,

settings: TweetServiceSettings,

timer: Timer

): () => Future[Unit] =

() =>

if (deciderGates.delayEraseUserTweets()) {

Future.sleep(settings.eraseUserTweetsDelay)(timer)

} else {

Future.Unit

}

}

object UndeleteTweetHandlerBuilder {

def apply(

tweetStorage: TweetStorageClient,

logicalRepos: LogicalRepositories,

tweetStores: TotalTweetStore,

parentUserIdRepo: ParentUserIdRepository.Type,

statsReceiver: StatsReceiver

): UndeleteTweetHandler.Type =

UndeleteTweetHandler(

undelete = tweetStorage.undelete,

tweetExists = tweetExists(tweetStorage),

getUser = FutureArrow(

UserRepository

.userGetter(

logicalRepos.optionalUserRepo,

UserQueryOptions(

// ExtendedProfile is needed to view a user's birthday to

// guarantee we are not undeleting tweets from when a user was < 13

TweetBuilder.userFields ++ Set(UserField.ExtendedProfile),

UserVisibility.All,

filteredAsFailure = false

)

)

.compose(UserKey.byId)

),

getDeletedTweets = tweetStorage.getDeletedTweets,

parentUserIdRepo = parentUserIdRepo,

save = save(

logicalRepos,

tweetStores,

statsReceiver

)

)

private def tweetExists(tweetStorage: TweetStorageClient): FutureArrow[TweetId, Boolean] =

FutureArrow { id =>

Stitch

.run(tweetStorage.getTweet(id))

.map {

case \_: GetTweet.Response.Found => true

case \_ => false

}

}

// 1. hydrates the undeleted tweet

// 2. hands a UndeleteTweetEvent to relevant stores.

// 3. return the hydrated tweet

def save(

logicalRepos: LogicalRepositories,

tweetStores: TotalTweetStore,

statsReceiver: StatsReceiver

): FutureArrow[UndeleteTweet.Event, Tweet] = {

val hydrateTweet =

WritePathHydration.hydrateTweet(

logicalRepos.tweetHydrators.hydrator,

statsReceiver.scope("undelete\_tweet")

)

val hydrateQuotedTweet =

WritePathHydration.hydrateQuotedTweet(

logicalRepos.optionalTweetRepo,

logicalRepos.optionalUserRepo,

logicalRepos.quoterHasAlreadyQuotedRepo

)

val hydrateUndeleteEvent =

WritePathHydration.hydrateUndeleteTweetEvent(

hydrateTweet = hydrateTweet,

hydrateQuotedTweet = hydrateQuotedTweet

)

FutureArrow[UndeleteTweet.Event, Tweet] { event =>

for {

hydratedEvent <- hydrateUndeleteEvent(event)

\_ <- tweetStores.undeleteTweet(hydratedEvent)

} yield hydratedEvent.tweet

}

}

}