package com.twitter.tweetypie

package config

import com.twitter.abdecider.ABDeciderFactory

import com.twitter.config.yaml.YamlConfig

import com.twitter.decider.Decider

import com.twitter.featureswitches.v2.FeatureSwitches

import com.twitter.finagle.memcached

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.servo.cache.\_

import com.twitter.servo.cache.{KeyValueResult => \_}

import com.twitter.servo.repository.\_

import com.twitter.stitch.NotFound

import com.twitter.stitch.Stitch

import com.twitter.stitch.repo.Repo

import com.twitter.stitch.timelineservice.TimelineService

import com.twitter.strato.client.{Client => StratoClient}

import com.twitter.stringcenter.client.ExternalStringRegistry

import com.twitter.stringcenter.client.MultiProjectStringCenter

import com.twitter.translation.Languages

import com.twitter.translation.YamlConfigLanguages

import com.twitter.tweetypie.caching.CacheOperations

import com.twitter.tweetypie.caching.Expiry

import com.twitter.tweetypie.caching.ServoCachedValueSerializer

import com.twitter.tweetypie.caching.StitchCaching

import com.twitter.tweetypie.caching.ValueSerializer

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

import com.twitter.tweetypie.core.FilteredState

import com.twitter.tweetypie.core.TweetResult

import com.twitter.tweetypie.hydrator.TextRepairer

import com.twitter.tweetypie.hydrator.TweetHydration

import com.twitter.tweetypie.hydrator.TweetQueryOptionsExpander

import com.twitter.tweetypie.repository.TweetRepository

import com.twitter.tweetypie.repository.UserRepository

import com.twitter.tweetypie.repository.\_

import com.twitter.tweetypie.serverutil.BoringStackTrace

import com.twitter.tweetypie.serverutil.ExceptionCounter

import com.twitter.tweetypie.thriftscala.DeviceSource

import com.twitter.tweetypie.thriftscala.Place

import com.twitter.tweetypie.thriftscala.entities.EntityExtractor

import com.twitter.tweetypie.util.StitchUtils

import com.twitter.util.Duration

import com.twitter.util.FuturePool

import com.twitter.util.Timer

import com.twitter.visibility.VisibilityLibrary

import com.twitter.visibility.common.KeywordMatcher

import com.twitter.visibility.common.LocalizationSource

import com.twitter.visibility.common.TweetMediaMetadataSource

import com.twitter.visibility.common.TweetPerspectiveSource

import com.twitter.visibility.common.UserRelationshipSource

import com.twitter.visibility.common.UserSource

import com.twitter.visibility.common.tflock.UserIsInvitedToConversationRepository

import com.twitter.visibility.configapi.configs.VisibilityDeciderGates

import com.twitter.visibility.generators.CountryNameGenerator

import com.twitter.visibility.generators.LocalizedInterstitialGenerator

import com.twitter.visibility.generators.TombstoneGenerator

import com.twitter.visibility.interfaces.tweets.DeletedTweetVisibilityLibrary

import com.twitter.visibility.interfaces.tweets.QuotedTweetVisibilityLibrary

import com.twitter.visibility.interfaces.tweets.TweetVisibilityLibrary

import com.twitter.visibility.interfaces.tweets.UserUnavailableStateVisibilityLibrary

import com.twitter.visibility.util.DeciderUtil

import com.twitter.visibility.util.FeatureSwitchUtil

import java.util.concurrent.Executors

/\*\*

\* LogicalRepositories is a layer above ExternalRepositories. These repos may have additional

\* logic layered in, such as memcache-caching, hot-key caching, etc. There may

\* also be multiple logical repositories mapped to an single external repository.

\*

\* These repositories are used in tweet hydration and tweet creation.

\*/

trait LogicalRepositories {

def card2Repo: Card2Repository.Type

def cardRepo: CardRepository.Type

def cardUsersRepo: CardUsersRepository.Type

def conversationIdRepo: ConversationIdRepository.Type

def conversationControlRepo: ConversationControlRepository.Type

def conversationMutedRepo: ConversationMutedRepository.Type

def containerAsGetTweetResultRepo: CreativesContainerMaterializationRepository.GetTweetType

def containerAsGetTweetFieldsResultRepo: CreativesContainerMaterializationRepository.GetTweetFieldsType

def deviceSourceRepo: DeviceSourceRepository.Type

def escherbirdAnnotationRepo: EscherbirdAnnotationRepository.Type

def geoScrubTimestampRepo: GeoScrubTimestampRepository.Type

def languageRepo: LanguageRepository.Type

def mediaMetadataRepo: MediaMetadataRepository.Type

def pastedMediaRepo: PastedMediaRepository.Type

def perspectiveRepo: PerspectiveRepository.Type

def placeRepo: PlaceRepository.Type

def profileGeoRepo: ProfileGeoRepository.Type

def quoterHasAlreadyQuotedRepo: QuoterHasAlreadyQuotedRepository.Type

def lastQuoteOfQuoterRepo: LastQuoteOfQuoterRepository.Type

def relationshipRepo: RelationshipRepository.Type

def stratoSafetyLabelsRepo: StratoSafetyLabelsRepository.Type

def stratoCommunityMembershipRepo: StratoCommunityMembershipRepository.Type

def stratoCommunityAccessRepo: StratoCommunityAccessRepository.Type

def stratoSuperFollowEligibleRepo: StratoSuperFollowEligibleRepository.Type

def stratoSuperFollowRelationsRepo: StratoSuperFollowRelationsRepository.Type

def stratoPromotedTweetRepo: StratoPromotedTweetRepository.Type

def stratoSubscriptionVerificationRepo: StratoSubscriptionVerificationRepository.Type

def takedownRepo: UserTakedownRepository.Type

def tweetSpamCheckRepo: TweetSpamCheckRepository.Type

def retweetSpamCheckRepo: RetweetSpamCheckRepository.Type

def tweetCountsRepo: TweetCountsRepository.Type

def tweetVisibilityRepo: TweetVisibilityRepository.Type

def quotedTweetVisibilityRepo: QuotedTweetVisibilityRepository.Type

def deletedTweetVisibilityRepo: DeletedTweetVisibilityRepository.Type

def unmentionedEntitiesRepo: UnmentionedEntitiesRepository.Type

def urlRepo: UrlRepository.Type

def userRepo: UserRepository.Type

def optionalUserRepo: UserRepository.Optional

def userIdentityRepo: UserIdentityRepository.Type

def userIsInvitedToConversationRepo: UserIsInvitedToConversationRepository.Type

def userProtectionRepo: UserProtectionRepository.Type

def userViewRepo: UserViewRepository.Type

def userVisibilityRepo: UserVisibilityRepository.Type

def tweetResultRepo: TweetResultRepository.Type

def tweetRepo: TweetRepository.Type

def optionalTweetRepo: TweetRepository.Optional

/\*\*

\* Not actually repositories, but intimately intertwined.

\*/

def tweetHydrators: TweetHydrators

}

object LogicalRepositories {

/\*\*

\* Middleware is a function that takes a stitch repo and returns a new stitch repo.

\*/

type Middleware[K, V] = (K => Stitch[V]) => K => Stitch[V]

// Middleware2 is a function that takes a two-arg stitch repo and returns a new two-arg stitch repo.

type Middleware2[K, C, V] = ((K, C) => Stitch[V]) => ((K, C) => Stitch[V])

val exceptionLog: Logger = Logger(getClass)

// Converts a Middleware2 to a Middleware for use with withMiddleware.

def tupledMiddleware[K, C, V](middleware2: Middleware2[K, C, V]): Middleware[(K, C), V] =

repo => middleware2(Function.untupled(repo)).tupled

object ObserveStitch {

def apply[K, V](

repo: K => Stitch[V],

repoName: String,

stats: StatsReceiver

): K => Stitch[V] = {

val successCounter = stats.counter("success")

val notFoundCounter = stats.counter("not\_found")

val latencyStat = stats.stat("latency\_ms")

val exceptionCounter =

ExceptionCounter(

stats,

// don't count FilteredState exceptions

FilteredState.ignoringCategorizer(ExceptionCounter.defaultCategorizer)

)

(key: K) =>

StitchUtils.trackLatency(latencyStat, repo(key)).respond {

case Return(\_) => successCounter.incr()

case Throw(NotFound) => notFoundCounter.incr()

case Throw(t) =>

val message = s"$repoName: $key"

if (BoringStackTrace.isBoring(t)) {

exceptionLog.debug(message, t)

} else {

exceptionLog.warn(message, t)

}

exceptionCounter(t)

}

}

}

/\*\*

\* Add middleware to configure a repository. The stats receiver is

\* scoped for the currently-configured repository. The `toRepo` field

\* is the repository with some set of middleware applied. Each method

\* adds a new middleware to the current repo, and returns it as a

\* `RepoConfig`, allowing method chaining.

\*

\* Since each method call applies a new middleware, the final middleware is

\* the outermost middleware, and thus the one that sees the arguments

\* first.

\*/

class RepoConfig[K, V](

val toRepo: K => Stitch[V],

stats: StatsReceiver,

name: String,

memcachedClientWithInProcessCaching: memcached.Client) {

def withMiddleware(middleware: Middleware[K, V]): RepoConfig[K, V] =

new RepoConfig[K, V](middleware(toRepo), stats, name, memcachedClientWithInProcessCaching)

/\*\*

\* Wraps a repo with success/failure/latency stats tracking and logs

\* exceptions. This will be applied to every repository.

\*

\* @param repoName Used when logging exceptions thrown by the underlying repo.

\*/

def observe(repoName: String = s"${name}\_repo"): RepoConfig[K, V] = {

withMiddleware { repo => ObserveStitch[K, V](repo, repoName, stats) }

}

/\*\*

\* Use the supplied cache to wrap the repository with a read-through

\* caching layer.

\*/

def caching(

cache: LockingCache[K, Cached[V]],

partialHandler: CachedResult.PartialHandler[K, V],

maxCacheRequestSize: Int = Int.MaxValue

): RepoConfig[K, V] = {

val stitchLockingCache = StitchLockingCache(

underlying = cache,

picker = new PreferNewestCached[V],

maxRequestSize = maxCacheRequestSize

)

val handler: CachedResult.Handler[K, V] =

CachedResult.Handler(

CachedResult.PartialHandler.orElse(

partialHandler,

CachedResult.failuresAreDoNotCache

)

)

withMiddleware { repo =>

CacheStitch[K, K, V](

repo = repo,

cache = stitchLockingCache,

identity,

handler = handler,

cacheable = CacheStitch.cacheFoundAndNotFound

)

}

}

def newCaching(

keySerializer: K => String,

valueSerializer: ValueSerializer[Try[V]]

): RepoConfig[K, V] =

withMiddleware { repo =>

val logger = Logger(s"com.twitter.tweetypie.config.LogicalRepositories.$name")

val cacheOperations: CacheOperations[K, Try[V]] =

new CacheOperations(

keySerializer = keySerializer,

valueSerializer = valueSerializer,

memcachedClient = memcachedClientWithInProcessCaching,

statsReceiver = stats.scope("caching"),

logger = logger

)

val tryRepo: K => Stitch[Try[V]] = repo.andThen(\_.liftToTry)

val cachingTryRepo: K => Stitch[Try[V]] = new StitchCaching(cacheOperations, tryRepo)

cachingTryRepo.andThen(\_.lowerFromTry)

}

def toRepo2[K1, C](implicit tupleToK: ((K1, C)) <:< K): (K1, C) => Stitch[V] =

(k1, c) => toRepo(tupleToK((k1, c)))

}

def softTtlPartialHandler[K, V](

softTtl: Option[V] => Duration,

softTtlPerturbationFactor: Float = 0.05f

): CachedResult.PartialHandler[K, V] =

CachedResult

.softTtlExpiration[K, V](softTtl, CachedResult.randomExpiry(softTtlPerturbationFactor))

def apply(

settings: TweetServiceSettings,

stats: StatsReceiver,

timer: Timer,

deciderGates: TweetypieDeciderGates,

external: ExternalRepositories,

caches: Caches,

stratoClient: StratoClient,

hasMedia: Tweet => Boolean,

clientIdHelper: ClientIdHelper,

featureSwitchesWithoutExperiments: FeatureSwitches,

): LogicalRepositories = {

val repoStats = stats.scope("repositories")

def repoConfig[K, V](name: String, repo: K => Stitch[V]): RepoConfig[K, V] =

new RepoConfig[K, V](

name = name,

toRepo = repo,

stats = repoStats.scope(name),

memcachedClientWithInProcessCaching = caches.memcachedClientWithInProcessCaching)

def repo2Config[K, C, V](name: String, repo: (K, C) => Stitch[V]): RepoConfig[(K, C), V] =

repoConfig[(K, C), V](name, repo.tupled)

new LogicalRepositories {

// the final tweetResultRepo has a circular dependency, where it depends on hydrators

// that in turn depend on the tweetResultRepo, so we create a `tweetResultRepo` function

// that proxies to `var finalTweetResultRepo`, which gets set at the end of this block.

var finalTweetResultRepo: TweetResultRepository.Type = null

val tweetResultRepo: TweetResultRepository.Type =

(tweetId, opts) => finalTweetResultRepo(tweetId, opts)

val tweetRepo: TweetRepository.Type = TweetRepository.fromTweetResult(tweetResultRepo)

val optionalTweetRepo: TweetRepository.Optional = TweetRepository.optional(tweetRepo)

val userRepo: UserRepository.Type =

repo2Config(repo = external.userRepo, name = "user")

.observe()

.toRepo2

val optionalUserRepo: UserRepository.Optional = UserRepository.optional(userRepo)

private[this] val tweetVisibilityStatsReceiver: StatsReceiver =

repoStats.scope("tweet\_visibility\_library")

private[this] val userUnavailableVisibilityStatsReceiver: StatsReceiver =

repoStats.scope("user\_unavailable\_visibility\_library")

private[this] val quotedTweetVisibilityStatsReceiver: StatsReceiver =

repoStats.scope("quoted\_tweet\_visibility\_library")

private[this] val deletedTweetVisibilityStatsReceiver: StatsReceiver =

repoStats.scope("deleted\_tweet\_visibility\_library")

// TweetVisibilityLibrary still uses the old c.t.logging.Logger

private[this] val tweetVisibilityLogger =

com.twitter.logging.Logger("com.twitter.tweetypie.TweetVisibility")

private[this] val visibilityDecider: Decider = DeciderUtil.mkDecider(

deciderOverlayPath = settings.vfDeciderOverlayFilename,

useLocalDeciderOverrides = true)

private[this] val visibilityDeciderGates = VisibilityDeciderGates(visibilityDecider)

private[this] def visibilityLibrary(statsReceiver: StatsReceiver) = VisibilityLibrary

.Builder(

log = tweetVisibilityLogger,

statsReceiver = statsReceiver,

memoizeSafetyLevelParams = visibilityDeciderGates.enableMemoizeSafetyLevelParams

)

.withDecider(visibilityDecider)

.withDefaultABDecider(isLocal = false)

.withCaptureDebugStats(Gate.True)

.withEnableComposableActions(Gate.True)

.withEnableFailClosed(Gate.True)

.withEnableShortCircuiting(visibilityDeciderGates.enableShortCircuitingTVL)

.withSpecialLogging(visibilityDeciderGates.enableSpecialLogging)

.build()

def countryNameGenerator(statsReceiver: StatsReceiver) = {

// TweetVisibilityLibrary, DeletedTweetVisibilityLibrary, and

// UserUnavailableVisibilityLibrary do not evaluate any Rules

// that require the display of country names in copy

CountryNameGenerator.providesWithCustomMap(Map.empty, statsReceiver)

}

def tombstoneGenerator(

countryNameGenerator: CountryNameGenerator,

statsReceiver: StatsReceiver

) =

TombstoneGenerator(

visibilityLibrary(statsReceiver).visParams,

countryNameGenerator,

statsReceiver)

private[this] val userUnavailableVisibilityLibrary =

UserUnavailableStateVisibilityLibrary(

visibilityLibrary(userUnavailableVisibilityStatsReceiver),

visibilityDecider,

tombstoneGenerator(

countryNameGenerator(userUnavailableVisibilityStatsReceiver),

userUnavailableVisibilityStatsReceiver

),

LocalizedInterstitialGenerator(visibilityDecider, userUnavailableVisibilityStatsReceiver)

)

val userIdentityRepo: UserIdentityRepository.Type =

repoConfig(repo = UserIdentityRepository(userRepo), name = "user\_identity")

.observe()

.toRepo

val userProtectionRepo: UserProtectionRepository.Type =

repoConfig(repo = UserProtectionRepository(userRepo), name = "user\_protection")

.observe()

.toRepo

val userViewRepo: UserViewRepository.Type =

repoConfig(repo = UserViewRepository(userRepo), name = "user\_view")

.observe()

.toRepo

val userVisibilityRepo: UserVisibilityRepository.Type =

repoConfig(

repo = UserVisibilityRepository(userRepo, userUnavailableVisibilityLibrary),

name = "user\_visibility"

).observe().toRepo

val urlRepo: UrlRepository.Type =

repoConfig(repo = external.urlRepo, name = "url")

.observe()

.toRepo

val profileGeoRepo: ProfileGeoRepository.Type =

repoConfig(repo = external.profileGeoRepo, name = "profile\_geo")

.observe()

.toRepo

val quoterHasAlreadyQuotedRepo: QuoterHasAlreadyQuotedRepository.Type =

repo2Config(repo = external.quoterHasAlreadyQuotedRepo, name = "quoter\_has\_already\_quoted")

.observe()

.toRepo2

val lastQuoteOfQuoterRepo: LastQuoteOfQuoterRepository.Type =

repo2Config(repo = external.lastQuoteOfQuoterRepo, name = "last\_quote\_of\_quoter")

.observe()

.toRepo2

val mediaMetadataRepo: MediaMetadataRepository.Type =

repoConfig(repo = external.mediaMetadataRepo, name = "media\_metadata")

.observe()

.toRepo

val perspectiveRepo: PerspectiveRepository.Type =

repoConfig(repo = external.perspectiveRepo, name = "perspective")

.observe()

.toRepo

val conversationMutedRepo: ConversationMutedRepository.Type =

TimelineService.GetPerspectives.getConversationMuted(perspectiveRepo)

// Because observe is applied before caching, only cache misses

// (i.e. calls to the underlying repo) are observed.

// Note that `newCaching` has stats around cache hit/miss but `caching` does not.

val deviceSourceRepo: DeviceSourceRepository.Type =

repoConfig(repo = external.deviceSourceRepo, name = "device\_source")

.observe()

.newCaching(

keySerializer = appIdStr => DeviceSourceKey(appIdStr).toString,

valueSerializer = ServoCachedValueSerializer(

codec = DeviceSource,

expiry = Expiry.byAge(settings.deviceSourceMemcacheTtl),

softTtl = settings.deviceSourceMemcacheSoftTtl

)

)

.caching(

cache = caches.deviceSourceInProcessCache,

partialHandler = softTtlPartialHandler(\_ => settings.deviceSourceInProcessSoftTtl)

)

.toRepo

// Because observe is applied before caching, only cache misses

// (i.e. calls to the underlying repo) are observed

// Note that `newCaching` has stats around cache hit/miss but `caching` does not.

val placeRepo: PlaceRepository.Type =

repoConfig(repo = external.placeRepo, name = "place")

.observe()

.newCaching(

keySerializer = placeKey => placeKey.toString,

valueSerializer = ServoCachedValueSerializer(

codec = Place,

expiry = Expiry.byAge(settings.placeMemcacheTtl),

softTtl = settings.placeMemcacheSoftTtl

)

)

.toRepo

val cardRepo: CardRepository.Type =

repoConfig(repo = external.cardRepo, name = "cards")

.observe()

.toRepo

val card2Repo: Card2Repository.Type =

repo2Config(repo = external.card2Repo, name = "card2")

.observe()

.toRepo2

val cardUsersRepo: CardUsersRepository.Type =

repo2Config(repo = external.cardUsersRepo, name = "card\_users")

.observe()

.toRepo2

val relationshipRepo: RelationshipRepository.Type =

repoConfig(repo = external.relationshipRepo, name = "relationship")

.observe()

.toRepo

val conversationIdRepo: ConversationIdRepository.Type =

repoConfig(repo = external.conversationIdRepo, name = "conversation\_id")

.observe()

.toRepo

val conversationControlRepo: ConversationControlRepository.Type =

repo2Config(

repo = ConversationControlRepository(tweetRepo, stats.scope("conversation\_control")),

name = "conversation\_control"

).observe().toRepo2

val containerAsGetTweetResultRepo: CreativesContainerMaterializationRepository.GetTweetType =

repo2Config(

repo = external.containerAsTweetRepo,

name = "container\_as\_tweet"

).observe().toRepo2

val containerAsGetTweetFieldsResultRepo: CreativesContainerMaterializationRepository.GetTweetFieldsType =

repo2Config(

repo = external.containerAsTweetFieldsRepo,

name = "container\_as\_tweet\_fields"

).observe().toRepo2

val languageRepo: LanguageRepository.Type = {

val pool = FuturePool(Executors.newFixedThreadPool(settings.numPenguinThreads))

repoConfig(repo = PenguinLanguageRepository(pool), name = "language")

.observe()

.toRepo

}

// Because observe is applied before caching, only cache misses

// (i.e. calls to the underlying repo) are observed

// Note that `newCaching` has stats around cache hit/miss but `caching` does not.

val tweetCountsRepo: TweetCountsRepository.Type =

repoConfig(repo = external.tweetCountsRepo, name = "counts")

.observe()

.caching(

cache = caches.tweetCountsCache,

partialHandler = softTtlPartialHandler {

case Some(0) => settings.tweetCountsMemcacheZeroSoftTtl

case \_ => settings.tweetCountsMemcacheNonZeroSoftTtl

},

maxCacheRequestSize = settings.tweetCountsCacheChunkSize

)

.toRepo

val pastedMediaRepo: PastedMediaRepository.Type =

repo2Config(repo = PastedMediaRepository(tweetRepo), name = "pasted\_media")

.observe()

.toRepo2

val escherbirdAnnotationRepo: EscherbirdAnnotationRepository.Type =

repoConfig(repo = external.escherbirdAnnotationRepo, name = "escherbird\_annotations")

.observe()

.toRepo

val stratoSafetyLabelsRepo: StratoSafetyLabelsRepository.Type =

repo2Config(repo = external.stratoSafetyLabelsRepo, name = "strato\_safety\_labels")

.observe()

.toRepo2

val stratoCommunityMembershipRepo: StratoCommunityMembershipRepository.Type =

repoConfig(

repo = external.stratoCommunityMembershipRepo,

name = "strato\_community\_memberships")

.observe()

.toRepo

val stratoCommunityAccessRepo: StratoCommunityAccessRepository.Type =

repoConfig(repo = external.stratoCommunityAccessRepo, name = "strato\_community\_access")

.observe()

.toRepo

val stratoSuperFollowEligibleRepo: StratoSuperFollowEligibleRepository.Type =

repoConfig(

repo = external.stratoSuperFollowEligibleRepo,

name = "strato\_super\_follow\_eligible")

.observe()

.toRepo

val stratoSuperFollowRelationsRepo: StratoSuperFollowRelationsRepository.Type =

repo2Config(

repo = external.stratoSuperFollowRelationsRepo,

name = "strato\_super\_follow\_relations")

.observe()

.toRepo2

val stratoPromotedTweetRepo: StratoPromotedTweetRepository.Type =

repoConfig(repo = external.stratoPromotedTweetRepo, name = "strato\_promoted\_tweet")

.observe()

.toRepo

val stratoSubscriptionVerificationRepo: StratoSubscriptionVerificationRepository.Type =

repo2Config(

repo = external.stratoSubscriptionVerificationRepo,

name = "strato\_subscription\_verification")

.observe()

.toRepo2

val unmentionedEntitiesRepo: UnmentionedEntitiesRepository.Type =

repo2Config(repo = external.unmentionedEntitiesRepo, name = "unmentioned\_entities")

.observe()

.toRepo2

private[this] val userSource =

UserSource.fromRepo(

Repo { (k, \_) =>

val opts = UserQueryOptions(k.fields, UserVisibility.All)

userRepo(UserKey(k.id), opts)

}

)

private[this] val userRelationshipSource =

UserRelationshipSource.fromRepo(

Repo[UserRelationshipSource.Key, Unit, Boolean] { (key, \_) =>

relationshipRepo(

RelationshipKey(key.subjectId, key.objectId, key.relationship)

)

}

)

private[this] val tweetPerspectiveSource =

TweetPerspectiveSource.fromGetPerspectives(perspectiveRepo)

private[this] val tweetMediaMetadataSource =

TweetMediaMetadataSource.fromFunction(mediaMetadataRepo)

val userIsInvitedToConversationRepo: UserIsInvitedToConversationRepository.Type =

repo2Config(

repo = external.userIsInvitedToConversationRepo,

name = "user\_is\_invited\_to\_conversation")

.observe()

.toRepo2

private[this] val stringCenterClient: MultiProjectStringCenter = {

val stringCenterProjects = settings.flags.stringCenterProjects().toList

val languages: Languages = new YamlConfigLanguages(

new YamlConfig(settings.flags.languagesConfig()))

val loggingAbDecider = ABDeciderFactory("/usr/local/config/abdecider/abdecider.yml")

.withEnvironment("production")

.buildWithLogging()

MultiProjectStringCenter(

projects = stringCenterProjects,

defaultBundlePath = MultiProjectStringCenter.StandardDefaultBundlePath,

refreshingBundlePath = MultiProjectStringCenter.StandardRefreshingBundlePath,

refreshingInterval = MultiProjectStringCenter.StandardRefreshingInterval,

requireDefaultBundleExists = true,

languages = languages,

statsReceiver = tweetVisibilityStatsReceiver,

loggingABDecider = loggingAbDecider

)

}

private[this] val stringRegistry: ExternalStringRegistry = new ExternalStringRegistry()

private[this] val localizationSource: LocalizationSource =

LocalizationSource.fromMultiProjectStringCenterClient(stringCenterClient, stringRegistry)

val tweetVisibilityRepo: TweetVisibilityRepository.Type = {

val tweetVisibilityLibrary: TweetVisibilityLibrary.Type =

TweetVisibilityLibrary(

visibilityLibrary(tweetVisibilityStatsReceiver),

userSource = userSource,

userRelationshipSource = userRelationshipSource,

keywordMatcher = KeywordMatcher.defaultMatcher(stats),

stratoClient = stratoClient,

localizationSource = localizationSource,

decider = visibilityDecider,

invitedToConversationRepo = userIsInvitedToConversationRepo,

tweetPerspectiveSource = tweetPerspectiveSource,

tweetMediaMetadataSource = tweetMediaMetadataSource,

tombstoneGenerator = tombstoneGenerator(

countryNameGenerator(tweetVisibilityStatsReceiver),

tweetVisibilityStatsReceiver

),

interstitialGenerator =

LocalizedInterstitialGenerator(visibilityDecider, tweetVisibilityStatsReceiver),

limitedActionsFeatureSwitches =

FeatureSwitchUtil.mkLimitedActionsFeatureSwitches(tweetVisibilityStatsReceiver),

enableParityTest = deciderGates.tweetVisibilityLibraryEnableParityTest

)

val underlying =

TweetVisibilityRepository(

tweetVisibilityLibrary,

visibilityDeciderGates,

tweetVisibilityLogger,

repoStats.scope("tweet\_visibility\_repo")

)

repoConfig(repo = underlying, name = "tweet\_visibility")

.observe()

.toRepo

}

val quotedTweetVisibilityRepo: QuotedTweetVisibilityRepository.Type = {

val quotedTweetVisibilityLibrary: QuotedTweetVisibilityLibrary.Type =

QuotedTweetVisibilityLibrary(

visibilityLibrary(quotedTweetVisibilityStatsReceiver),

userSource = userSource,

userRelationshipSource = userRelationshipSource,

visibilityDecider,

userStateVisibilityLibrary = userUnavailableVisibilityLibrary,

enableVfFeatureHydration = deciderGates.enableVfFeatureHydrationInQuotedTweetVLShim

)

val underlying =

QuotedTweetVisibilityRepository(quotedTweetVisibilityLibrary, visibilityDeciderGates)

repoConfig(repo = underlying, name = "quoted\_tweet\_visibility")

.observe()

.toRepo

}

val deletedTweetVisibilityRepo: DeletedTweetVisibilityRepository.Type = {

val deletedTweetVisibilityLibrary: DeletedTweetVisibilityLibrary.Type =

DeletedTweetVisibilityLibrary(

visibilityLibrary(deletedTweetVisibilityStatsReceiver),

visibilityDecider,

tombstoneGenerator(

countryNameGenerator(deletedTweetVisibilityStatsReceiver),

deletedTweetVisibilityStatsReceiver

)

)

val underlying = DeletedTweetVisibilityRepository.apply(

deletedTweetVisibilityLibrary

)

repoConfig(repo = underlying, name = "deleted\_tweet\_visibility")

.observe()

.toRepo

}

val takedownRepo: UserTakedownRepository.Type =

repoConfig(repo = UserTakedownRepository(userRepo), name = "takedowns")

.observe()

.toRepo

val tweetSpamCheckRepo: TweetSpamCheckRepository.Type =

repo2Config(repo = external.tweetSpamCheckRepo, name = "tweet\_spam\_check")

.observe()

.toRepo2

val retweetSpamCheckRepo: RetweetSpamCheckRepository.Type =

repoConfig(repo = external.retweetSpamCheckRepo, name = "retweet\_spam\_check")

.observe()

.toRepo

// Because observe is applied before caching, only cache misses

// (i.e. calls to the underlying repo) are observed

// Note that `newCaching` has stats around cache hit/miss but `caching` does not.

val geoScrubTimestampRepo: GeoScrubTimestampRepository.Type =

repoConfig(repo = external.geoScrubTimestampRepo, name = "geo\_scrub")

.observe()

.caching(

cache = caches.geoScrubCache,

partialHandler = (\_ => None)

)

.toRepo

val tweetHydrators: TweetHydrators =

TweetHydrators(

stats = stats,

deciderGates = deciderGates,

repos = this,

tweetDataCache = caches.tweetDataCache,

hasMedia = hasMedia,

featureSwitchesWithoutExperiments = featureSwitchesWithoutExperiments,

clientIdHelper = clientIdHelper,

)

val queryOptionsExpander: TweetQueryOptionsExpander.Type =

TweetQueryOptionsExpander.threadLocalMemoize(

TweetQueryOptionsExpander.expandDependencies

)

// mutations to tweets that we only need to apply when reading from the external

// repository, and not when reading from cache

val tweetMutation: Mutation[Tweet] =

Mutation

.all(

Seq(

EntityExtractor.mutationAll,

TextRepairer.BlankLineCollapser,

TextRepairer.CoreTextBugPatcher

)

).onlyIf(\_.coreData.isDefined)

val cachingTweetRepo: TweetResultRepository.Type =

repo2Config(repo = external.tweetResultRepo, name = "saved\_tweet")

.observe()

.withMiddleware { repo =>

// applies tweetMutation to the results of TweetResultRepository

val mutateResult = TweetResult.mutate(tweetMutation)

repo.andThen(stitchResult => stitchResult.map(mutateResult))

}

.withMiddleware(

tupledMiddleware(

CachingTweetRepository(

caches.tweetResultCache,

settings.tweetTombstoneTtl,

stats.scope("saved\_tweet", "cache"),

clientIdHelper,

deciderGates.logCacheExceptions,

)

)

)

.toRepo2

finalTweetResultRepo = repo2Config(repo = cachingTweetRepo, name = "tweet")

.withMiddleware(

tupledMiddleware(

TweetHydration.hydrateRepo(

tweetHydrators.hydrator,

tweetHydrators.cacheChangesEffect,

queryOptionsExpander

)

)

)

.observe()

.withMiddleware(tupledMiddleware(TweetResultRepository.shortCircuitInvalidIds))

.toRepo2

}

}

}