package com.twitter.home\_mixer.module

import com.google.inject.Provides

import com.twitter.conversions.DurationOps.\_

import com.twitter.conversions.PercentOps.\_

import com.twitter.finagle.mtls.authentication.ServiceIdentifier

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.thrift.ClientId

import com.twitter.graph\_feature\_service.{thriftscala => gfs}

import com.twitter.home\_mixer.param.HomeMixerInjectionNames.EarlybirdRepository

import com.twitter.home\_mixer.param.HomeMixerInjectionNames.GraphTwoHopRepository

import com.twitter.home\_mixer.param.HomeMixerInjectionNames.InterestsThriftServiceClient

import com.twitter.home\_mixer.param.HomeMixerInjectionNames.TweetypieContentRepository

import com.twitter.home\_mixer.param.HomeMixerInjectionNames.UserFollowedTopicIdsRepository

import com.twitter.home\_mixer.param.HomeMixerInjectionNames.UtegSocialProofRepository

import com.twitter.home\_mixer.util.earlybird.EarlybirdRequestUtil

import com.twitter.home\_mixer.util.tweetypie.RequestFields

import com.twitter.inject.TwitterModule

import com.twitter.interests.{thriftscala => int}

import com.twitter.product\_mixer.shared\_library.memcached\_client.MemcachedClientBuilder

import com.twitter.product\_mixer.shared\_library.thrift\_client.FinagleThriftClientBuilder

import com.twitter.product\_mixer.shared\_library.thrift\_client.Idempotent

import com.twitter.recos.recos\_common.{thriftscala => rc}

import com.twitter.recos.user\_tweet\_entity\_graph.{thriftscala => uteg}

import com.twitter.search.earlybird.{thriftscala => eb}

import com.twitter.servo.cache.Cached

import com.twitter.servo.cache.CachedSerializer

import com.twitter.servo.cache.FinagleMemcacheFactory

import com.twitter.servo.cache.MemcacheCacheFactory

import com.twitter.servo.cache.NonLockingCache

import com.twitter.servo.cache.ThriftSerializer

import com.twitter.servo.keyvalue.KeyValueResultBuilder

import com.twitter.servo.repository.CachingKeyValueRepository

import com.twitter.servo.repository.ChunkingStrategy

import com.twitter.servo.repository.KeyValueRepository

import com.twitter.servo.repository.KeyValueResult

import com.twitter.servo.repository.keysAsQuery

import com.twitter.spam.rtf.{thriftscala => sp}

import com.twitter.tweetypie.{thriftscala => tp}

import com.twitter.util.Future

import com.twitter.util.Return

import javax.inject.Named

import javax.inject.Singleton

import org.apache.thrift.protocol.TCompactProtocol

object ThriftFeatureRepositoryModule extends TwitterModule {

private val DefaultRPCChunkSize = 50

private val GFSInteractionIdsLimit = 10

type EarlybirdQuery = (Seq[Long], Long)

type UtegQuery = (Seq[Long], (Long, Map[Long, Double]))

@Provides

@Singleton

@Named(InterestsThriftServiceClient)

def providesInterestsThriftServiceClient(

clientId: ClientId,

serviceIdentifier: ServiceIdentifier,

statsReceiver: StatsReceiver

): int.InterestsThriftService.MethodPerEndpoint = {

FinagleThriftClientBuilder

.buildFinagleMethodPerEndpoint[

int.InterestsThriftService.ServicePerEndpoint,

int.InterestsThriftService.MethodPerEndpoint](

serviceIdentifier = serviceIdentifier,

clientId = clientId,

dest = "/s/interests-thrift-service/interests-thrift-service",

label = "interests",

statsReceiver = statsReceiver,

idempotency = Idempotent(1.percent),

timeoutPerRequest = 350.milliseconds,

timeoutTotal = 350.milliseconds

)

}

@Provides

@Singleton

@Named(UserFollowedTopicIdsRepository)

def providesUserFollowedTopicIdsRepository(

@Named(InterestsThriftServiceClient) client: int.InterestsThriftService.MethodPerEndpoint

): KeyValueRepository[Seq[Long], Long, Seq[Long]] = {

val lookupContext = Some(

int.ExplicitInterestLookupContext(Some(Seq(int.InterestRelationType.Followed)))

)

def lookup(userId: Long): Future[Seq[Long]] = {

client.getUserExplicitInterests(userId, lookupContext).map { interests =>

interests.flatMap {

\_.interestId match {

case int.InterestId.SemanticCore(semanticCoreInterest) => Some(semanticCoreInterest.id)

case \_ => None

}

}

}

}

val keyValueRepository = toRepository(lookup)

keyValueRepository

}

@Provides

@Singleton

@Named(UtegSocialProofRepository)

def providesUtegSocialProofRepository(

clientId: ClientId,

serviceIdentifier: ServiceIdentifier,

statsReceiver: StatsReceiver

): KeyValueRepository[UtegQuery, Long, uteg.TweetRecommendation] = {

val client = FinagleThriftClientBuilder.buildFinagleMethodPerEndpoint[

uteg.UserTweetEntityGraph.ServicePerEndpoint,

uteg.UserTweetEntityGraph.MethodPerEndpoint](

serviceIdentifier = serviceIdentifier,

clientId = clientId,

dest = "/s/cassowary/user\_tweet\_entity\_graph",

label = "uteg-social-proof-repo",

statsReceiver = statsReceiver,

idempotency = Idempotent(1.percent),

timeoutPerRequest = 150.milliseconds,

timeoutTotal = 250.milliseconds

)

val utegSocialProofTypes = Seq(

rc.SocialProofType.Favorite,

rc.SocialProofType.Retweet,

rc.SocialProofType.Reply

)

def lookup(

tweetIds: Seq[Long],

view: (Long, Map[Long, Double])

): Future[Seq[Option[uteg.TweetRecommendation]]] = {

val (userId, seedsWithWeights) = view

val socialProofRequest = uteg.SocialProofRequest(

requesterId = Some(userId),

seedsWithWeights = seedsWithWeights,

inputTweets = tweetIds,

socialProofTypes = Some(utegSocialProofTypes)

)

client.findTweetSocialProofs(socialProofRequest).map { result =>

val resultMap = result.socialProofResults.map(t => t.tweetId -> t).toMap

tweetIds.map(resultMap.get)

}

}

toRepositoryBatchWithView(lookup, chunkSize = 200)

}

@Provides

@Singleton

@Named(TweetypieContentRepository)

def providesTweetypieContentRepository(

clientId: ClientId,

serviceIdentifier: ServiceIdentifier,

statsReceiver: StatsReceiver

): KeyValueRepository[Seq[Long], Long, tp.Tweet] = {

val client = FinagleThriftClientBuilder

.buildFinagleMethodPerEndpoint[

tp.TweetService.ServicePerEndpoint,

tp.TweetService.MethodPerEndpoint](

serviceIdentifier = serviceIdentifier,

clientId = clientId,

dest = "/s/tweetypie/tweetypie",

label = "tweetypie-content-repo",

statsReceiver = statsReceiver,

idempotency = Idempotent(1.percent),

timeoutPerRequest = 300.milliseconds,

timeoutTotal = 500.milliseconds

)

def lookup(tweetIds: Seq[Long]): Future[Seq[Option[tp.Tweet]]] = {

val getTweetFieldsOptions = tp.GetTweetFieldsOptions(

tweetIncludes = RequestFields.ContentFields,

includeRetweetedTweet = false,

includeQuotedTweet = false,

forUserId = None,

safetyLevel = Some(sp.SafetyLevel.FilterNone),

visibilityPolicy = tp.TweetVisibilityPolicy.NoFiltering

)

val request = tp.GetTweetFieldsRequest(tweetIds = tweetIds, options = getTweetFieldsOptions)

client.getTweetFields(request).map { results =>

results.map {

case tp.GetTweetFieldsResult(\_, tp.TweetFieldsResultState.Found(found), \_, \_) =>

Some(found.tweet)

case \_ => None

}

}

}

val keyValueRepository = toRepositoryBatch(lookup, chunkSize = 20)

val cacheClient = MemcachedClientBuilder.buildRawMemcachedClient(

numTries = 1,

numConnections = 1,

requestTimeout = 200.milliseconds,

globalTimeout = 200.milliseconds,

connectTimeout = 200.milliseconds,

acquisitionTimeout = 200.milliseconds,

serviceIdentifier = serviceIdentifier,

statsReceiver = statsReceiver

)

val finagleMemcacheFactory =

FinagleMemcacheFactory(cacheClient, "/s/cache/home\_content\_features:twemcaches")

val cacheValueTransformer =

new ThriftSerializer[tp.Tweet](tp.Tweet, new TCompactProtocol.Factory())

val cachedSerializer = CachedSerializer.binary(cacheValueTransformer)

val cache = MemcacheCacheFactory(

memcache = finagleMemcacheFactory(),

ttl = 48.hours

)[Long, Cached[tp.Tweet]](cachedSerializer)

val lockingCache = new NonLockingCache(cache)

val cachedKeyValueRepository = new CachingKeyValueRepository(

keyValueRepository,

lockingCache,

keysAsQuery[Long]

)

cachedKeyValueRepository

}

@Provides

@Singleton

@Named(GraphTwoHopRepository)

def providesGraphTwoHopRepository(

clientId: ClientId,

serviceIdentifier: ServiceIdentifier,

statsReceiver: StatsReceiver

): KeyValueRepository[(Seq[Long], Long), Long, Seq[gfs.IntersectionValue]] = {

val client = FinagleThriftClientBuilder

.buildFinagleMethodPerEndpoint[gfs.Server.ServicePerEndpoint, gfs.Server.MethodPerEndpoint](

serviceIdentifier = serviceIdentifier,

clientId = clientId,

dest = "/s/cassowary/graph\_feature\_service-server",

label = "gfs-repo",

statsReceiver = statsReceiver,

idempotency = Idempotent(1.percent),

timeoutPerRequest = 350.milliseconds,

timeoutTotal = 500.milliseconds

)

def lookup(

userIds: Seq[Long],

viewerId: Long

): Future[Seq[Option[Seq[gfs.IntersectionValue]]]] = {

val gfsIntersectionRequest = gfs.GfsPresetIntersectionRequest(

userId = viewerId,

candidateUserIds = userIds,

presetFeatureTypes = gfs.PresetFeatureTypes.HtlTwoHop,

intersectionIdLimit = Some(GFSInteractionIdsLimit)

)

client

.getPresetIntersection(gfsIntersectionRequest)

.map { graphFeatureServiceResponse =>

val resultMap = graphFeatureServiceResponse.results

.map(result => result.candidateUserId -> result.intersectionValues).toMap

userIds.map(resultMap.get(\_))

}

}

toRepositoryBatchWithView(lookup, chunkSize = 200)

}

@Provides

@Singleton

@Named(EarlybirdRepository)

def providesEarlybirdSearchRepository(

client: eb.EarlybirdService.MethodPerEndpoint,

clientId: ClientId

): KeyValueRepository[EarlybirdQuery, Long, eb.ThriftSearchResult] = {

def lookup(

tweetIds: Seq[Long],

viewerId: Long

): Future[Seq[Option[eb.ThriftSearchResult]]] = {

val request = EarlybirdRequestUtil.getTweetsFeaturesRequest(

userId = Some(viewerId),

tweetIds = Some(tweetIds),

clientId = Some(clientId.name),

authorScoreMap = None,

tensorflowModel = Some("timelines\_rectweet\_replica")

)

client

.search(request).map { response =>

val resultMap = response.searchResults

.map(\_.results.map { result => result.id -> result }.toMap).getOrElse(Map.empty)

tweetIds.map(resultMap.get)

}

}

toRepositoryBatchWithView(lookup)

}

protected def toRepository[K, V](

hydrate: K => Future[V]

): KeyValueRepository[Seq[K], K, V] = {

def asRepository(keys: Seq[K]): Future[KeyValueResult[K, V]] = {

Future.collect(keys.map(hydrate(\_).liftToTry)).map { results =>

keys

.zip(results)

.foldLeft(new KeyValueResultBuilder[K, V]()) {

case (bldr, (k, result)) =>

result match {

case Return(v) => bldr.addFound(k, v)

case \_ => bldr.addNotFound(k)

}

}.result

}

}

asRepository

}

protected def toRepositoryBatch[K, V](

hydrate: Seq[K] => Future[Seq[Option[V]]],

chunkSize: Int = DefaultRPCChunkSize

): KeyValueRepository[Seq[K], K, V] = {

def repository(keys: Seq[K]): Future[KeyValueResult[K, V]] =

batchRepositoryProcess(keys, hydrate(keys))

KeyValueRepository.chunked(repository, ChunkingStrategy.equalSize(chunkSize))

}

protected def toRepositoryBatchWithView[K, T, V](

hydrate: (Seq[K], T) => Future[Seq[Option[V]]],

chunkSize: Int = DefaultRPCChunkSize

): KeyValueRepository[(Seq[K], T), K, V] = {

def repository(input: (Seq[K], T)): Future[KeyValueResult[K, V]] = {

val (keys, view) = input

batchRepositoryProcess(keys, hydrate(keys, view))

}

KeyValueRepository.chunked(repository, CustomChunkingStrategy.equalSizeWithView(chunkSize))

}

private def batchRepositoryProcess[K, V](

keys: Seq[K],

f: Future[Seq[Option[V]]]

): Future[KeyValueResult[K, V]] = {

f.liftToTry

.map {

case Return(values) =>

keys

.zip(values)

.foldLeft(new KeyValueResultBuilder[K, V]()) {

case (bldr, (k, value)) =>

value match {

case Some(v) => bldr.addFound(k, v)

case \_ => bldr.addNotFound(k)

}

}.result

case \_ =>

keys

.foldLeft(new KeyValueResultBuilder[K, V]()) {

case (bldr, k) => bldr.addNotFound(k)

}.result

}

}

// Use only for cases not already covered by Servo's [[ChunkingStrategy]]

object CustomChunkingStrategy {

def equalSizeWithView[K, T](maxSize: Int): ((Seq[K], T)) => Seq[(Seq[K], T)] = {

case (keys, view) =>

ChunkingStrategy

.equalSize[K](maxSize)(keys)

.map { chunk: Seq[K] => (chunk, view) }

}

}

}