package com.twitter.frigate.pushservice.config

import com.twitter.abuse.detection.scoring.thriftscala.TweetScoringRequest

import com.twitter.abuse.detection.scoring.thriftscala.TweetScoringResponse

import com.twitter.audience\_rewards.thriftscala.HasSuperFollowingRelationshipRequest

import com.twitter.bijection.scrooge.BinaryScalaCodec

import com.twitter.bijection.scrooge.CompactScalaCodec

import com.twitter.channels.common.thriftscala.ApiList

import com.twitter.channels.common.thriftscala.ApiListDisplayLocation

import com.twitter.channels.common.thriftscala.ApiListView

import com.twitter.content\_mixer.thriftscala.ContentMixer

import com.twitter.conversions.DurationOps.\_

import com.twitter.cortex.deepbird.thriftjava.DeepbirdPredictionService

import com.twitter.cr\_mixer.thriftscala.CrMixer

import com.twitter.datatools.entityservice.entities.sports.thriftscala.BaseballGameLiveUpdate

import com.twitter.datatools.entityservice.entities.sports.thriftscala.BasketballGameLiveUpdate

import com.twitter.datatools.entityservice.entities.sports.thriftscala.CricketMatchLiveUpdate

import com.twitter.datatools.entityservice.entities.sports.thriftscala.NflFootballGameLiveUpdate

import com.twitter.datatools.entityservice.entities.sports.thriftscala.SoccerMatchLiveUpdate

import com.twitter.discovery.common.configapi.ConfigParamsBuilder

import com.twitter.discovery.common.configapi.FeatureContextBuilder

import com.twitter.discovery.common.environment.{Environment => NotifEnvironment}

import com.twitter.escherbird.common.thriftscala.Domains

import com.twitter.escherbird.common.thriftscala.QualifiedId

import com.twitter.escherbird.metadata.thriftscala.EntityMegadata

import com.twitter.escherbird.metadata.thriftscala.MetadataService

import com.twitter.escherbird.util.metadatastitch.MetadataStitchClient

import com.twitter.escherbird.util.uttclient

import com.twitter.escherbird.util.uttclient.CacheConfigV2

import com.twitter.escherbird.util.uttclient.CachedUttClientV2

import com.twitter.escherbird.utt.strato.thriftscala.Environment

import com.twitter.eventbus.client.EventBusPublisherBuilder

import com.twitter.events.recos.thriftscala.EventsRecosService

import com.twitter.explore\_ranker.thriftscala.ExploreRanker

import com.twitter.featureswitches.v2.FeatureSwitches

import com.twitter.finagle.Memcached

import com.twitter.finagle.ThriftMux

import com.twitter.finagle.client.BackupRequestFilter

import com.twitter.finagle.client.ClientRegistry

import com.twitter.finagle.loadbalancer.Balancers

import com.twitter.finagle.memcached.Client

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

import com.twitter.finagle.mtls.client.MtlsStackClient.\_

import com.twitter.finagle.mux.transport.OpportunisticTls

import com.twitter.finagle.service.Retries

import com.twitter.finagle.service.RetryPolicy

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.thrift.ClientId

import com.twitter.finagle.thrift.RichClientParam

import com.twitter.finagle.util.DefaultTimer

import com.twitter.flockdb.client.\_

import com.twitter.flockdb.client.thriftscala.FlockDB

import com.twitter.frigate.common.base.RandomRanker

import com.twitter.frigate.common.candidate.\_

import com.twitter.frigate.common.config.RateLimiterGenerator

import com.twitter.frigate.common.entity\_graph\_client.RecommendedTweetEntitiesStore

import com.twitter.frigate.common.filter.DynamicRequestMeterFilter

import com.twitter.frigate.common.history.\_

import com.twitter.frigate.common.ml.feature.\_

import com.twitter.frigate.common.store.\_

import com.twitter.frigate.common.store.deviceinfo.DeviceInfoStore

import com.twitter.frigate.common.store.deviceinfo.MobileSdkStore

import com.twitter.frigate.common.store.interests.\_

import com.twitter.frigate.common.store.strato.StratoFetchableStore

import com.twitter.frigate.common.store.strato.StratoScannableStore

import com.twitter.frigate.common.util.Finagle.readOnlyThriftService

import com.twitter.frigate.common.util.\_

import com.twitter.frigate.data\_pipeline.features\_common.FeatureStoreUtil

import com.twitter.frigate.data\_pipeline.features\_common.\_

import com.twitter.frigate.data\_pipeline.thriftscala.UserHistoryKey

import com.twitter.frigate.data\_pipeline.thriftscala.UserHistoryValue

import com.twitter.frigate.dau\_model.thriftscala.DauProbability

import com.twitter.frigate.magic\_events.thriftscala.FanoutEvent

import com.twitter.frigate.pushcap.thriftscala.PushcapUserHistory

import com.twitter.frigate.pushservice.model.PushTypes.PushCandidate

import com.twitter.frigate.pushservice.model.PushTypes.Target

import com.twitter.frigate.pushservice.adaptor.LoggedOutPushCandidateSourceGenerator

import com.twitter.frigate.pushservice.adaptor.PushCandidateSourceGenerator

import com.twitter.frigate.pushservice.config.mlconfig.DeepbirdV2ModelConfig

import com.twitter.frigate.pushservice.ml.\_

import com.twitter.frigate.pushservice.params.\_

import com.twitter.frigate.pushservice.rank.LoggedOutRanker

import com.twitter.frigate.pushservice.rank.RFPHLightRanker

import com.twitter.frigate.pushservice.rank.RFPHRanker

import com.twitter.frigate.pushservice.rank.SubscriptionCreatorRanker

import com.twitter.frigate.pushservice.refresh\_handler.\_

import com.twitter.frigate.pushservice.refresh\_handler.cross.CandidateCopyExpansion

import com.twitter.frigate.pushservice.send\_handler.SendHandlerPushCandidateHydrator

import com.twitter.frigate.pushservice.store.\_

import com.twitter.frigate.pushservice.take.CandidateNotifier

import com.twitter.frigate.pushservice.take.NotificationSender

import com.twitter.frigate.pushservice.take.NotificationServiceRequest

import com.twitter.frigate.pushservice.take.NotificationServiceSender

import com.twitter.frigate.pushservice.take.NtabOnlyChannelSelector

import com.twitter.frigate.pushservice.take.history.EventBusWriter

import com.twitter.frigate.pushservice.take.history.HistoryWriter

import com.twitter.frigate.pushservice.take.sender.Ibis2Sender

import com.twitter.frigate.pushservice.take.sender.NtabSender

import com.twitter.frigate.pushservice.take.LoggedOutRefreshForPushNotifier

import com.twitter.frigate.pushservice.util.RFPHTakeStepUtil

import com.twitter.frigate.pushservice.util.SendHandlerPredicateUtil

import com.twitter.frigate.scribe.thriftscala.NotificationScribe

import com.twitter.frigate.thriftscala.\_

import com.twitter.frigate.user\_states.thriftscala.MRUserHmmState

import com.twitter.geoduck.backend.hydration.thriftscala.Hydration

import com.twitter.geoduck.common.thriftscala.PlaceQueryFields

import com.twitter.geoduck.common.thriftscala.PlaceType

import com.twitter.geoduck.common.thriftscala.{Location => GeoLocation}

import com.twitter.geoduck.service.common.clientmodules.GeoduckUserLocate

import com.twitter.geoduck.service.common.clientmodules.GeoduckUserLocateModule

import com.twitter.geoduck.service.thriftscala.LocationResponse

import com.twitter.geoduck.thriftscala.LocationService

import com.twitter.gizmoduck.context.thriftscala.ReadConfig

import com.twitter.gizmoduck.context.thriftscala.TestUserConfig

import com.twitter.gizmoduck.testusers.client.TestUserClientBuilder

import com.twitter.gizmoduck.thriftscala.LookupContext

import com.twitter.gizmoduck.thriftscala.QueryFields

import com.twitter.gizmoduck.thriftscala.User

import com.twitter.gizmoduck.thriftscala.UserService

import com.twitter.hermit.pop\_geo.thriftscala.PopTweetsInPlace

import com.twitter.hermit.predicate.socialgraph.SocialGraphPredicate

import com.twitter.hermit.predicate.tweetypie.PerspectiveReadableStore

import com.twitter.hermit.store.\_

import com.twitter.hermit.store.common.\_

import com.twitter.hermit.store.gizmoduck.GizmoduckUserStore

import com.twitter.hermit.store.metastore.UserCountryStore

import com.twitter.hermit.store.metastore.UserLanguagesStore

import com.twitter.hermit.store.scarecrow.ScarecrowCheckEventStore

import com.twitter.hermit.store.semantic\_core.MetaDataReadableStore

import com.twitter.hermit.store.semantic\_core.SemanticEntityForQuery

import com.twitter.hermit.store.timezone.GizmoduckUserUtcOffsetStore

import com.twitter.hermit.store.timezone.UtcOffsetStore

import com.twitter.hermit.store.tweetypie.TweetyPieStore

import com.twitter.hermit.store.tweetypie.UserTweet

import com.twitter.hermit.store.user\_htl\_session\_store.UserHTLLastVisitReadableStore

import com.twitter.hermit.stp.thriftscala.STPResult

import com.twitter.hss.api.thriftscala.UserHealthSignal

import com.twitter.hss.api.thriftscala.UserHealthSignal.\_

import com.twitter.hss.api.thriftscala.UserHealthSignalResponse

import com.twitter.interests.thriftscala.InterestId

import com.twitter.interests.thriftscala.InterestsThriftService

import com.twitter.interests.thriftscala.{UserInterests => Interests}

import com.twitter.interests\_discovery.thriftscala.InterestsDiscoveryService

import com.twitter.interests\_discovery.thriftscala.NonPersonalizedRecommendedLists

import com.twitter.interests\_discovery.thriftscala.RecommendedListsRequest

import com.twitter.interests\_discovery.thriftscala.RecommendedListsResponse

import com.twitter.kujaku.domain.thriftscala.MachineTranslationResponse

import com.twitter.livevideo.timeline.client.v2.LiveVideoTimelineClient

import com.twitter.livevideo.timeline.domain.v2.{Event => LiveEvent}

import com.twitter.livevideo.timeline.thrift.thriftscala.TimelineService

import com.twitter.logging.Logger

import com.twitter.ml.api.thriftscala.{DataRecord => ThriftDataRecord}

import com.twitter.ml.featurestore.catalog.entities.core.{Author => TweetAuthorEntity}

import com.twitter.ml.featurestore.catalog.entities.core.{User => TargetUserEntity}

import com.twitter.ml.featurestore.catalog.entities.core.{UserAuthor => UserAuthorEntity}

import com.twitter.ml.featurestore.catalog.entities.magicrecs.{SocialContext => SocialContextEntity}

import com.twitter.ml.featurestore.catalog.entities.magicrecs.{UserSocialContext => TargetUserSocialContextEntity}

import com.twitter.ml.featurestore.timelines.thriftscala.TimelineScorerScoreView

import com.twitter.notificationservice.api.thriftscala.DeleteCurrentTimelineForUserRequest

import com.twitter.notificationservice.genericfeedbackstore.FeedbackPromptValue

import com.twitter.notificationservice.genericfeedbackstore.GenericFeedbackStore

import com.twitter.notificationservice.genericfeedbackstore.GenericFeedbackStoreBuilder

import com.twitter.notificationservice.scribe.manhattan.FeedbackSignalManhattanClient

import com.twitter.notificationservice.scribe.manhattan.GenericNotificationsFeedbackRequest

import com.twitter.notificationservice.thriftscala.CaretFeedbackDetails

import com.twitter.notificationservice.thriftscala.CreateGenericNotificationRequest

import com.twitter.notificationservice.thriftscala.CreateGenericNotificationResponse

import com.twitter.notificationservice.thriftscala.DeleteGenericNotificationRequest

import com.twitter.notificationservice.thriftscala.GenericNotificationOverrideKey

import com.twitter.notificationservice.thriftscala.NotificationService$FinagleClient

import com.twitter.nrel.heavyranker.CandidateFeatureHydrator

import com.twitter.nrel.heavyranker.FeatureHydrator

import com.twitter.nrel.heavyranker.{PushPredictionServiceStore => RelevancePushPredictionServiceStore}

import com.twitter.nrel.heavyranker.{TargetFeatureHydrator => RelevanceTargetFeatureHydrator}

import com.twitter.nrel.lightranker.MagicRecsServeDataRecordLightRanker

import com.twitter.nrel.lightranker.{Config => LightRankerConfig}

import com.twitter.onboarding.task.service.thriftscala.FatigueFlowEnrollment

import com.twitter.periscope.api.thriftscala.AudioSpacesLookupContext

import com.twitter.permissions\_storage.thriftscala.AppPermission

import com.twitter.recommendation.interests.discovery.core.config.{DeployConfig => InterestDeployConfig}

import com.twitter.recommendation.interests.discovery.popgeo.deploy.PopGeoInterestProvider

import com.twitter.recos.user\_tweet\_entity\_graph.thriftscala.UserTweetEntityGraph

import com.twitter.recos.user\_user\_graph.thriftscala.UserUserGraph

import com.twitter.rux.common.strato.thriftscala.UserTargetingProperty

import com.twitter.scio.nsfw\_user\_segmentation.thriftscala.NSFWProducer

import com.twitter.scio.nsfw\_user\_segmentation.thriftscala.NSFWUserSegmentation

import com.twitter.search.earlybird.thriftscala.EarlybirdService

import com.twitter.service.gen.scarecrow.thriftscala.ScarecrowService

import com.twitter.service.metastore.gen.thriftscala.Location

import com.twitter.simclusters\_v2.thriftscala.SimClustersInferredEntities

import com.twitter.socialgraph.thriftscala.SocialGraphService

import com.twitter.spam.rtf.thriftscala.SafetyLevel

import com.twitter.stitch.tweetypie.TweetyPie.TweetyPieResult

import com.twitter.storage.client.manhattan.kv.Guarantee

import com.twitter.storage.client.manhattan.kv.ManhattanKVClient

import com.twitter.storage.client.manhattan.kv.ManhattanKVClientMtlsParams

import com.twitter.storage.client.manhattan.kv.ManhattanKVEndpoint

import com.twitter.storage.client.manhattan.kv.ManhattanKVEndpointBuilder

import com.twitter.storehaus.ReadableStore

import com.twitter.storehaus\_internal.manhattan.Apollo

import com.twitter.storehaus\_internal.manhattan.Athena

import com.twitter.storehaus\_internal.manhattan.Dataset

import com.twitter.storehaus\_internal.manhattan.ManhattanStore

import com.twitter.storehaus\_internal.manhattan.Nash

import com.twitter.storehaus\_internal.manhattan.Omega

import com.twitter.storehaus\_internal.memcache.MemcacheStore

import com.twitter.storehaus\_internal.util.ClientName

import com.twitter.storehaus\_internal.util.ZkEndPoint

import com.twitter.strato.catalog.Scan.Slice

import com.twitter.strato.client.Strato

import com.twitter.strato.client.UserId

import com.twitter.strato.columns.frigate.logged\_out\_web\_notifications.thriftscala.LOWebNotificationMetadata

import com.twitter.strato.columns.notifications.thriftscala.SourceDestUserRequest

import com.twitter.strato.generated.client.geo.user.FrequentSoftUserLocationClientColumn

import com.twitter.strato.generated.client.ml.featureStore.TimelineScorerTweetScoresV1ClientColumn

import com.twitter.strato.generated.client.notifications.space\_device\_follow\_impl.SpaceDeviceFollowingClientColumn

import com.twitter.strato.generated.client.periscope.CoreOnAudioSpaceClientColumn

import com.twitter.strato.generated.client.periscope.ParticipantsOnAudioSpaceClientColumn

import com.twitter.strato.generated.client.rux.TargetingPropertyOnUserClientColumn

import com.twitter.strato.generated.client.socialgraph.graphs.creatorSubscriptionTimeline.{CountEdgesBySourceClientColumn => CreatorSubscriptionNumTweetsColumn}

import com.twitter.strato.generated.client.translation.service.IsTweetTranslatableClientColumn

import com.twitter.strato.generated.client.translation.service.platform.MachineTranslateTweetClientColumn

import com.twitter.strato.generated.client.trends.trip.TripTweetsAirflowProdClientColumn

import com.twitter.strato.thrift.ScroogeConvImplicits.\_

import com.twitter.taxi.common.AppId

import com.twitter.taxi.deploy.Cluster

import com.twitter.taxi.deploy.Env

import com.twitter.topiclisting.TopicListing

import com.twitter.topiclisting.TopicListingBuilder

import com.twitter.trends.trip\_v1.trip\_tweets.thriftscala.TripDomain

import com.twitter.trends.trip\_v1.trip\_tweets.thriftscala.TripTweets

import com.twitter.tsp.thriftscala.TopicSocialProofRequest

import com.twitter.tsp.thriftscala.TopicSocialProofResponse

import com.twitter.tweetypie.thriftscala.GetTweetOptions

import com.twitter.tweetypie.thriftscala.Tweet.VisibleTextRangeField

import com.twitter.tweetypie.thriftscala.TweetService

import com.twitter.ubs.thriftscala.AudioSpace

import com.twitter.ubs.thriftscala.Participants

import com.twitter.ubs.thriftscala.SellerApplicationState

import com.twitter.user\_session\_store.thriftscala.UserSession

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.Timer

import com.twitter.util.tunable.TunableMap

import com.twitter.wtf.scalding.common.thriftscala.UserFeatures

import org.apache.thrift.protocol.TCompactProtocol

import com.twitter.timelinescorer.thriftscala.v1.ScoredTweet

import com.twitter.ubs.thriftscala.SellerTrack

import com.twitter.wtf.candidate.thriftscala.CandidateSeq

trait DeployConfig extends Config {

// Any finagle clients should not be defined as lazy. If defined lazy,

// ClientRegistry.expAllRegisteredClientsResolved() call in init will not ensure that the clients

// are active before thrift endpoint is active. We want the clients to be active, because zookeeper

// resolution triggered by first request(s) might result in the request(s) failing.

def serviceIdentifier: ServiceIdentifier

def tunableMap: TunableMap

def featureSwitches: FeatureSwitches

override val isProd: Boolean =

serviceIdentifier.environment == PushConstants.ServiceProdEnvironmentName

def shardParams: ShardParams

def log: Logger

implicit def statsReceiver: StatsReceiver

implicit val timer: Timer = DefaultTimer

def notifierThriftClientId: ClientId

def loggedOutNotifierThriftClientId: ClientId

def pushserviceThriftClientId: ClientId

def deepbirdv2PredictionServiceDest: String

def featureStoreUtil: FeatureStoreUtil

def targetLevelFeaturesConfig: PushFeaturesConfig

private val manhattanClientMtlsParams = ManhattanKVClientMtlsParams(

serviceIdentifier = serviceIdentifier,

opportunisticTls = OpportunisticTls.Required

)

// Commonly used clients

val gizmoduckClient = {

val client = ThriftMux.client

.withMutualTls(serviceIdentifier)

.withClientId(pushserviceThriftClientId)

.build[UserService.MethodPerEndpoint](

dest = "/s/gizmoduck/gizmoduck"

)

/\*\*

\* RequestContext test user config to allow reading test user accounts on pushservice for load

\* testing

\*/

val GizmoduckTestUserConfig = TestUserConfig(

clientId = Some(pushserviceThriftClientId.name),

readConfig = Some(ReadConfig(includeTestUsers = true))

)

TestUserClientBuilder[UserService.MethodPerEndpoint]

.withClient(client)

.withConfig(GizmoduckTestUserConfig)

.build()

}

val sgsClient = {

val service = readOnlyThriftService(

"",

"/s/socialgraph/socialgraph",

statsReceiver,

pushserviceThriftClientId,

mTLSServiceIdentifier = Some(serviceIdentifier)

)

new SocialGraphService.FinagledClient(service)

}

val tweetyPieClient = {

val service = readOnlyThriftService(

"",

"/s/tweetypie/tweetypie",

statsReceiver,

notifierThriftClientId,

mTLSServiceIdentifier = Some(serviceIdentifier)

)

new TweetService.FinagledClient(service)

}

lazy val geoduckHydrationClient: Hydration.MethodPerEndpoint = {

val servicePerEndpoint = ThriftMux.client

.withLabel("geoduck\_hydration")

.withClientId(pushserviceThriftClientId)

.withMutualTls(serviceIdentifier)

.methodBuilder("/s/geo/hydration")

.withTimeoutPerRequest(10.seconds)

.withTimeoutTotal(10.seconds)

.idempotent(maxExtraLoad = 0.0)

.servicePerEndpoint[Hydration.ServicePerEndpoint]

Hydration.MethodPerEndpoint(servicePerEndpoint)

}

lazy val geoduckLocationClient: LocationService.MethodPerEndpoint = {

val servicePerEndpoint = ThriftMux.client

.withLabel("geoduck\_location")

.withClientId(pushserviceThriftClientId)

.withMutualTls(serviceIdentifier)

.methodBuilder("/s/geo/geoduck\_locationservice")

.withTimeoutPerRequest(10.seconds)

.withTimeoutTotal(10.seconds)

.idempotent(maxExtraLoad = 0.0)

.servicePerEndpoint[LocationService.ServicePerEndpoint]

LocationService.MethodPerEndpoint(servicePerEndpoint)

}

override val geoDuckV2Store: ReadableStore[Long, LocationResponse] = {

val geoduckLocate: GeoduckUserLocate = GeoduckUserLocateModule.providesGeoduckUserLocate(

locationServiceClient = geoduckLocationClient,

hydrationClient = geoduckHydrationClient,

unscopedStatsReceiver = statsReceiver

)

val store: ReadableStore[Long, LocationResponse] = ReadableStore

.convert[GeoduckRequest, Long, LocationResponse, LocationResponse](

GeoduckStoreV2(geoduckLocate))({ userId: Long =>

GeoduckRequest(

userId,

placeTypes = Set(

PlaceType.City,

PlaceType.Metro,

PlaceType.Country,

PlaceType.ZipCode,

PlaceType.Admin0,

PlaceType.Admin1),

placeFields = Set(PlaceQueryFields.PlaceNames),

includeCountryCode = true

)

})({ locationResponse: LocationResponse => Future.value(locationResponse) })

val \_cacheName = "geoduckv2\_in\_memory\_cache"

ObservedCachedReadableStore.from(

store,

ttl = 20.seconds,

maxKeys = 1000,

cacheName = \_cacheName,

windowSize = 10000L

)(statsReceiver.scope(\_cacheName))

}

private val deepbirdServiceBase = ThriftMux.client

.withClientId(pushserviceThriftClientId)

.withMutualTls(serviceIdentifier)

.withLoadBalancer(Balancers.p2c())

.newService(deepbirdv2PredictionServiceDest, "DeepbirdV2PredictionService")

val deepbirdPredictionServiceClient = new DeepbirdPredictionService.ServiceToClient(

Finagle

.retryReadFilter(

tries = 3,

statsReceiver = statsReceiver.scope("DeepbirdV2PredictionService"))

.andThen(Finagle.timeoutFilter(timeout = 10.seconds))

.andThen(deepbirdServiceBase),

RichClientParam(serviceName = "DeepbirdV2PredictionService", clientStats = statsReceiver)

)

val manhattanStarbuckAppId = "frigate\_pushservice\_starbuck"

val metastoreLocationAppId = "frigate\_notifier\_metastore\_location"

val manhattanMetastoreAppId = "frigate\_pushservice\_penguin"

def pushServiceMHCacheDest: String

def pushServiceCoreSvcsCacheDest: String

def poptartImpressionsCacheDest: String = "/srv#/prod/local/cache/poptart\_impressions"

def entityGraphCacheDest: String

val pushServiceCacheClient: Client = MemcacheStore.memcachedClient(

name = ClientName("memcache-pushservice"),

dest = ZkEndPoint(pushServiceMHCacheDest),

statsReceiver = statsReceiver,

timeout = 2.seconds,

serviceIdentifier = serviceIdentifier

)

val pushServiceCoreSvcsCacheClient: Client =

MemcacheStore.memcachedClient(

name = ClientName("memcache-pushservice-core-svcs"),

dest = ZkEndPoint(pushServiceCoreSvcsCacheDest),

statsReceiver = statsReceiver,

serviceIdentifier = serviceIdentifier,

timeout = 2.seconds,

)

val poptartImpressionsCacheClient: Client =

MemcacheStore.memcachedClient(

name = ClientName("memcache-pushservice-poptart-impressions"),

dest = ZkEndPoint(poptartImpressionsCacheDest),

statsReceiver = statsReceiver,

serviceIdentifier = serviceIdentifier,

timeout = 2.seconds

)

val entityGraphCacheClient: Client = MemcacheStore.memcachedClient(

name = ClientName("memcache-pushservice-entity-graph"),

dest = ZkEndPoint(entityGraphCacheDest),

statsReceiver = statsReceiver,

serviceIdentifier = serviceIdentifier,

timeout = 2.seconds

)

val stratoClient = {

val pushserviceThriftClient = ThriftMux.client.withClientId(pushserviceThriftClientId)

val baseBuilder = Strato

.Client(pushserviceThriftClient)

.withMutualTls(serviceIdentifier)

val finalBuilder = if (isServiceLocal) {

baseBuilder.withRequestTimeout(Duration.fromSeconds(15))

} else {

baseBuilder.withRequestTimeout(Duration.fromSeconds(3))

}

finalBuilder.build()

}

val interestThriftServiceClient = ThriftMux.client

.withClientId(pushserviceThriftClientId)

.withMutualTls(serviceIdentifier)

.withRequestTimeout(3.seconds)

.configured(Retries.Policy(RetryPolicy.tries(1)))

.configured(BackupRequestFilter.Configured(maxExtraLoad = 0.0, sendInterrupts = false))

.withStatsReceiver(statsReceiver)

.build[InterestsThriftService.MethodPerEndpoint](

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

label = "interests-lookup"

)

def memcacheCASDest: String

override val casLock: CasLock = {

val magicrecsCasMemcacheClient = Memcached.client

.withMutualTls(serviceIdentifier)

.withLabel("mr-cas-memcache-client")

.withRequestTimeout(3.seconds)

.withStatsReceiver(statsReceiver)

.configured(Retries.Policy(RetryPolicy.tries(3)))

.newTwemcacheClient(memcacheCASDest)

.withStrings

MemcacheCasLock(magicrecsCasMemcacheClient)

}

override val pushInfoStore: ReadableStore[Long, UserForPushTargeting] = {

StratoFetchableStore.withUnitView[Long, UserForPushTargeting](

stratoClient,

"frigate/magicrecs/pushRecsTargeting.User")

}

override val loggedOutPushInfoStore: ReadableStore[Long, LOWebNotificationMetadata] = {

StratoFetchableStore.withUnitView[Long, LOWebNotificationMetadata](

stratoClient,

"frigate/magicrecs/web/loggedOutWebUserStoreMh"

)

}

// Setting up model stores

override val dauProbabilityStore: ReadableStore[Long, DauProbability] = {

StratoFetchableStore

.withUnitView[Long, DauProbability](stratoClient, "frigate/magicrecs/dauProbability.User")

}

override val nsfwConsumerStore = {

StratoFetchableStore.withUnitView[Long, NSFWUserSegmentation](

stratoClient,

"frigate/nsfw-user-segmentation/nsfwUserSegmentation.User")

}

override val nsfwProducerStore = {

StratoFetchableStore.withUnitView[Long, NSFWProducer](

stratoClient,

"frigate/nsfw-user-segmentation/nsfwProducer.User"

)

}

override val idsStore: ReadableStore[RecommendedListsRequest, RecommendedListsResponse] = {

val service = Finagle.readOnlyThriftService(

name = "interests-discovery-service",

dest = "/s/interests\_discovery/interests\_discovery",

statsReceiver,

pushserviceThriftClientId,

requestTimeout = 4.seconds,

tries = 2,

mTLSServiceIdentifier = Some(serviceIdentifier)

)

val client = new InterestsDiscoveryService.FinagledClient(

service = service,

RichClientParam(serviceName = "interests-discovery-service")

)

InterestDiscoveryStore(client)

}

override val popGeoLists = {

StratoFetchableStore.withUnitView[String, NonPersonalizedRecommendedLists](

stratoClient,

column = "recommendations/interests\_discovery/recommendations\_mh/OrganicPopgeoLists"

)

}

override val listAPIStore = {

val fetcher = stratoClient

.fetcher[Long, ApiListView, ApiList]("channels/hydration/apiList.List")

StratoFetchableStore.withView[Long, ApiListView, ApiList](

fetcher,

ApiListView(ApiListDisplayLocation.Recommendations)

)

}

override val reactivatedUserInfoStore = {

val stratoFetchableStore = StratoFetchableStore

.withUnitView[Long, String](stratoClient, "ml/featureStore/recentReactivationTime.User")

ObservedReadableStore(

stratoFetchableStore

)(statsReceiver.scope("RecentReactivationTime"))

}

override val openedPushByHourAggregatedStore: ReadableStore[Long, Map[Int, Int]] = {

StratoFetchableStore

.withUnitView[Long, Map[Int, Int]](

stratoClient,

"frigate/magicrecs/opendPushByHourAggregated.User")

}

private val lexClient: LiveVideoTimelineClient = {

val lexService =

new TimelineService.FinagledClient(

readOnlyThriftService(

name = "lex",

dest = lexServiceDest,

statsReceiver = statsReceiver.scope("lex-service"),

thriftClientId = pushserviceThriftClientId,

requestTimeout = 5.seconds,

mTLSServiceIdentifier = Some(serviceIdentifier)

),

clientParam = RichClientParam(serviceName = "lex")

)

new LiveVideoTimelineClient(lexService)

}

override val lexServiceStore = {

ObservedCachedReadableStore.from[EventRequest, LiveEvent](

buildStore(LexServiceStore(lexClient), "lexServiceStore"),

ttl = 1.hour,

maxKeys = 1000,

cacheName = "lexServiceStore\_cache",

windowSize = 10000L

)(statsReceiver.scope("lexServiceStore\_cache"))

}

val inferredEntitiesFromInterestedInKeyedByClusterColumn =

"recommendations/simclusters\_v2/inferred\_entities/inferredEntitiesFromInterestedInKeyedByCluster"

override val simClusterToEntityStore: ReadableStore[Int, SimClustersInferredEntities] = {

val store = StratoFetchableStore

.withUnitView[Int, SimClustersInferredEntities](

stratoClient,

inferredEntitiesFromInterestedInKeyedByClusterColumn)

ObservedCachedReadableStore.from[Int, SimClustersInferredEntities](

buildStore(store, "simcluster\_entity\_store\_cache"),

ttl = 6.hours,

maxKeys = 1000,

cacheName = "simcluster\_entity\_store\_cache",

windowSize = 10000L

)(statsReceiver.scope("simcluster\_entity\_store\_cache"))

}

def fanoutMetadataColumn: String

override val fanoutMetadataStore: ReadableStore[(Long, Long), FanoutEvent] = {

val store = StratoFetchableStore

.withUnitView[(Long, Long), FanoutEvent](stratoClient, fanoutMetadataColumn)

ObservedCachedReadableStore.from[(Long, Long), FanoutEvent](

buildStore(store, "fanoutMetadataStore"),

ttl = 10.minutes,

maxKeys = 1000,

cacheName = "fanoutMetadataStore\_cache",

windowSize = 10000L

)(statsReceiver.scope("fanoutMetadataStore\_cache"))

}

/\*\*

\* PostRanking Feature Store Client

\*/

override def postRankingFeatureStoreClient = {

val clientStats = statsReceiver.scope("post\_ranking\_feature\_store\_client")

val clientConfig =

FeatureStoreClientBuilder.getClientConfig(PostRankingFeaturesConfig(), featureStoreUtil)

FeatureStoreClientBuilder.getDynamicFeatureStoreClient(clientConfig, clientStats)

}

/\*\*

\* Interests lookup store

\*/

override val interestsWithLookupContextStore = {

ObservedCachedReadableStore.from[InterestsLookupRequestWithContext, Interests](

buildStore(

new InterestsWithLookupContextStore(interestThriftServiceClient, statsReceiver),

"InterestsWithLookupContextStore"

),

ttl = 1.minute,

maxKeys = 1000,

cacheName = "interestsWithLookupContextStore\_cache",

windowSize = 10000L

)

}

/\*\*

\* OptOutInterestsStore

\*/

override lazy val optOutUserInterestsStore: ReadableStore[Long, Seq[InterestId]] = {

buildStore(

InterestsOptOutwithLookUpContextStore(interestThriftServiceClient),

"InterestsOptOutStore"

)

}

override val topicListing: TopicListing =

if (isServiceLocal) {

new TopicListingBuilder(statsReceiver.scope("topiclisting"), Some(localConfigRepoPath)).build

} else {

new TopicListingBuilder(statsReceiver.scope("topiclisting"), None).build

}

val cachedUttClient = {

val DefaultUttCacheConfig = CacheConfigV2(capacity = 100)

val uttClientCacheConfigs = uttclient.UttClientCacheConfigsV2(

DefaultUttCacheConfig,

DefaultUttCacheConfig,

DefaultUttCacheConfig,

DefaultUttCacheConfig

)

new CachedUttClientV2(stratoClient, Environment.Prod, uttClientCacheConfigs, statsReceiver)

}

override val uttEntityHydrationStore =

new UttEntityHydrationStore(cachedUttClient, statsReceiver, log)

private lazy val dbv2PredictionServiceScoreStore: RelevancePushPredictionServiceStore =

DeepbirdV2ModelConfig.buildPredictionServiceScoreStore(

deepbirdPredictionServiceClient,

"deepbirdv2\_magicrecs"

)

// Customized model to PredictionServiceStoreMap

// It is used to specify the predictionServiceStore for the models not in the default dbv2PredictionServiceScoreStore

private lazy val modelToPredictionServiceStoreMap: Map[

WeightedOpenOrNtabClickModel.ModelNameType,

RelevancePushPredictionServiceStore

] = Map()

override lazy val weightedOpenOrNtabClickModelScorer = new PushMLModelScorer(

PushMLModel.WeightedOpenOrNtabClickProbability,

modelToPredictionServiceStoreMap,

dbv2PredictionServiceScoreStore,

statsReceiver.scope("weighted\_oonc\_scoring")

)

override lazy val optoutModelScorer = new PushMLModelScorer(

PushMLModel.OptoutProbability,

Map.empty,

dbv2PredictionServiceScoreStore,

statsReceiver.scope("optout\_scoring")

)

override lazy val filteringModelScorer = new PushMLModelScorer(

PushMLModel.FilteringProbability,

Map.empty,

dbv2PredictionServiceScoreStore,

statsReceiver.scope("filtering\_scoring")

)

private val queryFields: Set[QueryFields] = Set(

QueryFields.Profile,

QueryFields.Account,

QueryFields.Roles,

QueryFields.Discoverability,

QueryFields.Safety,

QueryFields.Takedowns,

QueryFields.Labels,

QueryFields.Counts,

QueryFields.ExtendedProfile

)

// Setting up safeUserStore

override val safeUserStore =

// in-memory cache

ObservedCachedReadableStore.from[Long, User](

ObservedReadableStore(

GizmoduckUserStore.safeStore(

client = gizmoduckClient,

queryFields = queryFields,

safetyLevel = SafetyLevel.FilterNone,

statsReceiver = statsReceiver

)

)(statsReceiver.scope("SafeUserStore")),

ttl = 1.minute,

maxKeys = 5e4.toInt,

cacheName = "safeUserStore\_cache",

windowSize = 10000L

)(statsReceiver.scope("safeUserStore\_cache"))

val mobileSdkStore = MobileSdkStore(

"frigate\_mobile\_sdk\_version\_apollo",

"mobile\_sdk\_versions\_scalding",

manhattanClientMtlsParams,

Apollo

)

val deviceUserStore = ObservedReadableStore(

GizmoduckUserStore(

client = gizmoduckClient,

queryFields = Set(QueryFields.Devices),

context = LookupContext(includeSoftUsers = true),

statsReceiver = statsReceiver

)

)(statsReceiver.scope("devicesUserStore"))

override val deviceInfoStore = DeviceInfoStore(

ObservedMemcachedReadableStore.fromCacheClient(

backingStore = ObservedReadableStore(

mobileSdkStore

)(statsReceiver.scope("uncachedMobileSdkVersionsStore")),

cacheClient = pushServiceCacheClient,

ttl = 12.hours

)(

valueInjection = BinaryScalaCodec(SdkVersionValue),

statsReceiver = statsReceiver.scope("MobileSdkVersionsStore"),

keyToString = {

case SdkVersionKey(Some(userId), Some(clientId)) =>

s"DeviceInfoStore/$userId/$clientId"

case SdkVersionKey(Some(userId), None) => s"DeviceInfoStore/$userId/\_"

case SdkVersionKey(None, Some(clientId)) =>

s"DeviceInfoStore/\_/$clientId"

case SdkVersionKey(None, None) => s"DeviceInfoStore/\_"

}

),

deviceUserStore

)

// Setting up edgeStore

override val edgeStore = SocialGraphPredicate.buildEdgeStore(sgsClient)

override val socialGraphServiceProcessStore = SocialGraphServiceProcessStore(edgeStore)

def userTweetEntityGraphDest: String

def userUserGraphDest: String

def lexServiceDest: String

// Setting up the history store

def frigateHistoryCacheDest: String

val notificationHistoryStore: NotificationHistoryStore = {

val manhattanStackBasedClient = ThriftMux.client

.withClientId(notifierThriftClientId)

.withOpportunisticTls(OpportunisticTls.Required)

.withMutualTls(

serviceIdentifier

)

val manhattanHistoryMethodBuilder = manhattanStackBasedClient

.withLabel("manhattan\_history\_v2")

.withRequestTimeout(10.seconds)

.withStatsReceiver(statsReceiver)

.methodBuilder(Omega.wilyName)

.withMaxRetries(3)

NotificationHistoryStore.build(

"frigate\_notifier",

"frigate\_notifications\_v2",

manhattanHistoryMethodBuilder,

maxRetryCount = 3

)

}

val emailNotificationHistoryStore: ReadOnlyHistoryStore = {

val client = ManhattanKVClient(

appId = "frigate\_email\_history",

dest = "/s/manhattan/omega.native-thrift",

mtlsParams = ManhattanKVClientMtlsParams(

serviceIdentifier = serviceIdentifier,

opportunisticTls = OpportunisticTls.Required

)

)

val endpoint = ManhattanKVEndpointBuilder(client)

.defaultGuarantee(Guarantee.SoftDcReadMyWrites)

.statsReceiver(statsReceiver)

.build()

ReadOnlyHistoryStore(ManhattanKVHistoryStore(endpoint, dataset = "frigate\_email\_history"))(

statsReceiver)

}

val manhattanKVLoggedOutHistoryStoreEndpoint: ManhattanKVEndpoint = {

val mhClient = ManhattanKVClient(

"frigate\_notification\_logged\_out\_history",

Nash.wilyName,

manhattanClientMtlsParams)

ManhattanKVEndpointBuilder(mhClient)

.defaultGuarantee(Guarantee.SoftDcReadMyWrites)

.defaultMaxTimeout(5.seconds)

.maxRetryCount(3)

.statsReceiver(statsReceiver)

.build()

}

val manhattanKVNtabHistoryStoreEndpoint: ManhattanKVEndpoint = {

val mhClient = ManhattanKVClient("frigate\_ntab", Omega.wilyName, manhattanClientMtlsParams)

ManhattanKVEndpointBuilder(mhClient)

.defaultGuarantee(Guarantee.SoftDcReadMyWrites)

.defaultMaxTimeout(5.seconds)

.maxRetryCount(3)

.statsReceiver(statsReceiver)

.build()

}

val nTabHistoryStore: ReadableWritableStore[(Long, String), GenericNotificationOverrideKey] = {

ObservedReadableWritableStore(

NTabHistoryStore(manhattanKVNtabHistoryStoreEndpoint, "frigate\_ntab\_generic\_notif\_history")

)(statsReceiver.scope("NTabHistoryStore"))

}

override lazy val ocfFatigueStore: ReadableStore[OCFHistoryStoreKey, FatigueFlowEnrollment] =

new OCFPromptHistoryStore(

manhattanAppId = "frigate\_pushservice\_ocf\_fatigue\_store",

dataset = "fatigue\_v1",

manhattanClientMtlsParams

)

def historyStore: PushServiceHistoryStore

def emailHistoryStore: PushServiceHistoryStore

def loggedOutHistoryStore: PushServiceHistoryStore

override val hydratedLabeledPushRecsStore: ReadableStore[UserHistoryKey, UserHistoryValue] = {

val labeledHistoryMemcacheClient = {

MemcacheStore.memcachedClient(

name = ClientName("history-memcache"),

dest = ZkEndPoint(frigateHistoryCacheDest),

statsReceiver = statsReceiver,

timeout = 2.seconds,

serviceIdentifier = serviceIdentifier

)

}

implicit val keyCodec = CompactScalaCodec(UserHistoryKey)

implicit val valueCodec = CompactScalaCodec(UserHistoryValue)

val dataset: Dataset[UserHistoryKey, UserHistoryValue] =

Dataset(

"",

"frigate\_data\_pipeline\_pushservice",

"labeled\_push\_recs\_aggregated\_hydrated",

Athena

)

ObservedMemcachedReadableStore.fromCacheClient(

backingStore = ObservedReadableStore(buildManhattanStore(dataset))(

statsReceiver.scope("UncachedHydratedLabeledPushRecsStore")

),

cacheClient = labeledHistoryMemcacheClient,

ttl = 6.hours

)(

valueInjection = valueCodec,

statsReceiver = statsReceiver.scope("HydratedLabeledPushRecsStore"),

keyToString = {

case UserHistoryKey.UserId(userId) => s"HLPRS/$userId"

case unknownKey =>

throw new IllegalArgumentException(s"Unknown userHistoryStore cache key $unknownKey")

}

)

}

override val realTimeClientEventStore: RealTimeClientEventStore = {

val client = ManhattanKVClient(

"frigate\_eventstream",

"/s/manhattan/omega.native-thrift",

manhattanClientMtlsParams

)

val endpoint =

ManhattanKVEndpointBuilder(client)

.defaultGuarantee(Guarantee.SoftDcReadMyWrites)

.defaultMaxTimeout(3.seconds)

.statsReceiver(statsReceiver)

.build()

ManhattanRealTimeClientEventStore(endpoint, "realtime\_client\_events", statsReceiver, None)

}

override val onlineUserHistoryStore: ReadableStore[OnlineUserHistoryKey, UserHistoryValue] = {

OnlineUserHistoryStore(realTimeClientEventStore)

}

override val userMediaRepresentationStore = UserMediaRepresentationStore(

"user\_media\_representation",

"user\_media\_representation\_dataset",

manhattanClientMtlsParams

)

override val producerMediaRepresentationStore = ObservedMemcachedReadableStore.fromCacheClient(

backingStore = UserMediaRepresentationStore(

"user\_media\_representation",

"producer\_media\_representation\_dataset",

manhattanClientMtlsParams

)(statsReceiver.scope("UncachedProducerMediaRepStore")),

cacheClient = pushServiceCacheClient,

ttl = 4.hours

)(

valueInjection = BinaryScalaCodec(UserMediaRepresentation),

keyToString = { k: Long => s"ProducerMediaRepStore/$k" },

statsReceiver.scope("ProducerMediaRepStore")

)

override val mrUserStatePredictionStore = {

StratoFetchableStore.withUnitView[Long, MRUserHmmState](

stratoClient,

"frigate/magicrecs/mrUserStatePrediction.User")

}

override val userHTLLastVisitStore =

UserHTLLastVisitReadableStore(

"pushservice\_htl\_user\_session",

"tls\_user\_session\_store",

statsReceiver.scope("userHTLLastVisitStore"),

manhattanClientMtlsParams

)

val crMixerClient: CrMixer.MethodPerEndpoint = new CrMixer.FinagledClient(

readOnlyThriftService(

"cr-mixer",

"/s/cr-mixer/cr-mixer-plus",

statsReceiver,

pushserviceThriftClientId,

requestTimeout = 5.seconds,

mTLSServiceIdentifier = Some(serviceIdentifier)

),

clientParam = RichClientParam(serviceName = "cr-mixer")

)

val crMixerStore = CrMixerTweetStore(crMixerClient)(statsReceiver.scope("CrMixerTweetStore"))

val contentMixerClient: ContentMixer.MethodPerEndpoint = new ContentMixer.FinagledClient(

readOnlyThriftService(

"content-mixer",

"/s/corgi-shared/content-mixer",

statsReceiver,

pushserviceThriftClientId,

requestTimeout = 5.seconds,

mTLSServiceIdentifier = Some(serviceIdentifier)

),

clientParam = RichClientParam(serviceName = "content-mixer")

)

val exploreRankerClient: ExploreRanker.MethodPerEndpoint =

new ExploreRanker.FinagledClient(

readOnlyThriftService(

"explore-ranker",

"/s/explore-ranker/explore-ranker",

statsReceiver,

pushserviceThriftClientId,

requestTimeout = 5.seconds,

mTLSServiceIdentifier = Some(serviceIdentifier)

),

clientParam = RichClientParam(serviceName = "explore-ranker")

)

val contentMixerStore = {

ObservedReadableStore(ContentMixerStore(contentMixerClient))(

statsReceiver.scope("ContentMixerStore"))

}

val exploreRankerStore = {

ObservedReadableStore(ExploreRankerStore(exploreRankerClient))(

statsReceiver.scope("ExploreRankerStore")

)

}

val gizmoduckUtcOffsetStore = ObservedReadableStore(

GizmoduckUserUtcOffsetStore.fromUserStore(safeUserStore)

)(statsReceiver.scope("GizmoUserUtcOffsetStore"))

override val userUtcOffsetStore =

UtcOffsetStore

.makeMemcachedUtcOffsetStore(

gizmoduckUtcOffsetStore,

pushServiceCoreSvcsCacheClient,

ReadableStore.empty,

manhattanStarbuckAppId,

manhattanClientMtlsParams

)(statsReceiver)

.mapValues(Duration.fromSeconds)

override val cachedTweetyPieStoreV2 = {

val getTweetOptions = Some(

GetTweetOptions(

safetyLevel = Some(SafetyLevel.MagicRecsV2),

includeRetweetCount = true,

includeReplyCount = true,

includeFavoriteCount = true,

includeQuotedTweet = true,

additionalFieldIds = Seq(VisibleTextRangeField.id)

)

)

buildCachedTweetyPieStore(getTweetOptions, "tp\_v2")

}

override val cachedTweetyPieStoreV2NoVF = {

val getTweetOptions = Some(

GetTweetOptions(

safetyLevel = Some(SafetyLevel.FilterDefault),

includeRetweetCount = true,

includeReplyCount = true,

includeFavoriteCount = true,

includeQuotedTweet = true,

additionalFieldIds = Seq(VisibleTextRangeField.id),

)

)

buildCachedTweetyPieStore(getTweetOptions, "tp\_v2\_noVF")

}

override val safeCachedTweetyPieStoreV2 = {

val getTweetOptions = Some(

GetTweetOptions(

safetyLevel = Some(SafetyLevel.MagicRecsAggressiveV2),

includeRetweetCount = true,

includeReplyCount = true,

includeFavoriteCount = true,

includeQuotedTweet = true,

additionalFieldIds = Seq(VisibleTextRangeField.id)

)

)

buildCachedTweetyPieStore(getTweetOptions, "sftp\_v2")

}

override val userTweetTweetyPieStore: ReadableStore[UserTweet, TweetyPieResult] = {

val getTweetOptions = Some(

GetTweetOptions(

safetyLevel = Some(SafetyLevel.MagicRecsV2),

includeRetweetCount = true,

includeReplyCount = true,

includeFavoriteCount = true,

includeQuotedTweet = true,

additionalFieldIds = Seq(VisibleTextRangeField.id)

)

)

TweetyPieStore.buildUserTweetStore(

client = tweetyPieClient,

options = getTweetOptions

)

}

override val safeUserTweetTweetyPieStore: ReadableStore[UserTweet, TweetyPieResult] = {

val getTweetOptions = Some(

GetTweetOptions(

safetyLevel = Some(SafetyLevel.MagicRecsAggressiveV2),

includeRetweetCount = true,

includeReplyCount = true,

includeFavoriteCount = true,

includeQuotedTweet = true,

additionalFieldIds = Seq(VisibleTextRangeField.id)

)

)

TweetyPieStore.buildUserTweetStore(

client = tweetyPieClient,

options = getTweetOptions

)

}

override val tweetContentFeatureCacheStore: ReadableStore[Long, ThriftDataRecord] = {

ObservedMemcachedReadableStore.fromCacheClient(

backingStore = TweetContentFeatureReadableStore(stratoClient),

cacheClient = poptartImpressionsCacheClient,

ttl = 12.hours

)(

valueInjection = BinaryScalaCodec(ThriftDataRecord),

statsReceiver = statsReceiver.scope("TweetContentFeaturesCacheStore"),

keyToString = { k: Long => s"tcf/$k" }

)

}

lazy val tweetTranslationStore: ReadableStore[

TweetTranslationStore.Key,

TweetTranslationStore.Value

] = {

val isTweetTranslatableStore =

StratoFetchableStore

.withUnitView[IsTweetTranslatableClientColumn.Key, Boolean](

fetcher = new IsTweetTranslatableClientColumn(stratoClient).fetcher

)

val translateTweetStore =

StratoFetchableStore

.withUnitView[MachineTranslateTweetClientColumn.Key, MachineTranslationResponse](

fetcher = new MachineTranslateTweetClientColumn(stratoClient).fetcher

)

ObservedReadableStore(

TweetTranslationStore(translateTweetStore, isTweetTranslatableStore, statsReceiver)

)(statsReceiver.scope("tweetTranslationStore"))

}

val scarecrowClient = new ScarecrowService.FinagledClient(

readOnlyThriftService(

"",

"/s/abuse/scarecrow",

statsReceiver,

notifierThriftClientId,

requestTimeout = 5.second,

mTLSServiceIdentifier = Some(serviceIdentifier)

),

clientParam = RichClientParam(serviceName = "")

)

// Setting up scarecrow store

override val scarecrowCheckEventStore = {

ScarecrowCheckEventStore(scarecrowClient)

}

// setting up the perspective store

override val userTweetPerspectiveStore = {

val service = new DynamicRequestMeterFilter(

tunableMap(PushServiceTunableKeys.TweetPerspectiveStoreQpsLimit),

RateLimiterGenerator.asTuple(\_, shardParams.numShards, 40),

PushQPSLimitConstants.PerspectiveStoreQPS)(timer)

.andThen(

readOnlyThriftService(

"tweetypie\_perspective\_service",

"/s/tweetypie/tweetypie",

statsReceiver,

notifierThriftClientId,

mTLSServiceIdentifier = Some(serviceIdentifier)

)

)

val client = new TweetService.FinagledClient(

service,

clientParam = RichClientParam(serviceName = "tweetypie\_perspective\_client"))

ObservedReadableStore(

PerspectiveReadableStore(client)

)(statsReceiver.scope("TweetPerspectiveStore"))

}

//user country code store, used in RecsWithheldContentPredicate - wrapped by memcache based cache

override val userCountryStore =

ObservedMemcachedReadableStore.fromCacheClient(

backingStore = ObservedReadableStore(

UserCountryStore(metastoreLocationAppId, manhattanClientMtlsParams)

)(statsReceiver.scope("userCountryStore")),

cacheClient = pushServiceCacheClient,

ttl = 12.hours

)(

valueInjection = BinaryScalaCodec(Location),

statsReceiver = statsReceiver.scope("UserCountryStore"),

keyToString = { k: Long => s"UserCountryStore/$k" }

)

override val audioSpaceParticipantsStore: ReadableStore[String, Participants] = {

val store = StratoFetchableStore

.DefaultStratoFetchableStore(

fetcher = new ParticipantsOnAudioSpaceClientColumn(stratoClient).fetcher

).composeKeyMapping[String](broadcastId =>

(broadcastId, AudioSpacesLookupContext(forUserId = None)))

ObservedCachedReadableStore

.from(

store = buildStore(store, "AudioSpaceParticipantsStore"),

ttl = 20.seconds,

maxKeys = 200,

cacheName = "AudioSpaceParticipantsStore",

windowSize = 200

)

}

override val topicSocialProofServiceStore: ReadableStore[

TopicSocialProofRequest,

TopicSocialProofResponse

] = {

StratoFetchableStore.withUnitView[TopicSocialProofRequest, TopicSocialProofResponse](

stratoClient,

"topic-signals/tsp/topic-social-proof")

}

override val spaceDeviceFollowStore: ReadableStore[SourceDestUserRequest, Boolean] = {

StratoFetchableStore.withUnitView(

fetcher = new SpaceDeviceFollowingClientColumn(stratoClient).fetcher

)

}

override val audioSpaceStore: ReadableStore[String, AudioSpace] = {

val store = StratoFetchableStore

.DefaultStratoFetchableStore(

fetcher = new CoreOnAudioSpaceClientColumn(stratoClient).fetcher

).composeKeyMapping[String] { broadcastId =>

(broadcastId, AudioSpacesLookupContext(forUserId = None))

}

ObservedCachedReadableStore

.from(

store = buildStore(store, "AudioSpaceVisibilityStore"),

ttl = 1.minute,

maxKeys = 5000,

cacheName = "AudioSpaceVisibilityStore",

windowSize = 10000L)

}

override val userLanguagesStore = UserLanguagesStore(

manhattanMetastoreAppId,

manhattanClientMtlsParams,

statsReceiver.scope("user\_languages\_store")

)

val tflockClient: TFlockClient = new TFlockClient(

new FlockDB.FinagledClient(

readOnlyThriftService(

"tflockClient",

"/s/tflock/tflock",

statsReceiver,

pushserviceThriftClientId,

mTLSServiceIdentifier = Some(serviceIdentifier)

),

serviceName = "tflock",

stats = statsReceiver

),

defaultPageSize = 1000

)

val rawFlockClient = ThriftMux.client

.withClientId(pushserviceThriftClientId)

.withMutualTls(serviceIdentifier)

.build[FlockDB.MethodPerEndpoint]("/s/flock/flock")

val flockClient: FlockClient = new FlockClient(

rawFlockClient,

defaultPageSize = 100

)

override val recentFollowsStore: FlockFollowStore = {

val dStats = statsReceiver.scope("FlockRecentFollowsStore")

FlockFollowStore(flockClient, dStats)

}

def notificationServiceClient: NotificationService$FinagleClient

def notificationServiceSend(

target: Target,

request: CreateGenericNotificationRequest

): Future[CreateGenericNotificationResponse]

def notificationServiceDelete(

request: DeleteGenericNotificationRequest

): Future[Unit]

def notificationServiceDeleteTimeline(

request: DeleteCurrentTimelineForUserRequest

): Future[Unit]

override val notificationServiceSender: ReadableStore[

NotificationServiceRequest,

CreateGenericNotificationResponse

] = {

new NotificationServiceSender(

notificationServiceSend,

PushParams.EnableWritesToNotificationServiceParam,

PushParams.EnableWritesToNotificationServiceForAllEmployeesParam,

PushParams.EnableWritesToNotificationServiceForEveryoneParam

)

}

val eventRecosServiceClient = {

val dest = "/s/events-recos/events-recos-service"

new EventsRecosService.FinagledClient(

readOnlyThriftService(

"EventRecosService",

dest,

statsReceiver,

pushserviceThriftClientId,

mTLSServiceIdentifier = Some(serviceIdentifier)

),

clientParam = RichClientParam(serviceName = "EventRecosService")

)

}

lazy val recommendedTrendsCandidateSource = RecommendedTrendsCandidateSource(

TrendsRecommendationStore(eventRecosServiceClient, statsReceiver))

override val softUserGeoLocationStore: ReadableStore[Long, GeoLocation] =

StratoFetchableStore.withUnitView[Long, GeoLocation](fetcher =

new FrequentSoftUserLocationClientColumn(stratoClient).fetcher)

lazy val candidateSourceGenerator = new PushCandidateSourceGenerator(

earlybirdCandidateSource,

userTweetEntityGraphCandidates,

cachedTweetyPieStoreV2,

safeCachedTweetyPieStoreV2,

userTweetTweetyPieStore,

safeUserTweetTweetyPieStore,

cachedTweetyPieStoreV2NoVF,

edgeStore,

interestsWithLookupContextStore,

uttEntityHydrationStore,

geoDuckV2Store,

topTweetsByGeoStore,

topTweetsByGeoV2VersionedStore,

ruxTweetImpressionsStore,

recommendedTrendsCandidateSource,

recentTweetsByAuthorsStore,

topicSocialProofServiceStore,

crMixerStore,

contentMixerStore,

exploreRankerStore,

softUserGeoLocationStore,

tripTweetCandidateStore,

popGeoLists,

idsStore

)

lazy val loCandidateSourceGenerator = new LoggedOutPushCandidateSourceGenerator(

tripTweetCandidateStore,

geoDuckV2Store,

safeCachedTweetyPieStoreV2,

cachedTweetyPieStoreV2NoVF,

cachedTweetyPieStoreV2,

contentMixerStore,

softUserGeoLocationStore,

topTweetsByGeoStore,

topTweetsByGeoV2VersionedStore

)

lazy val rfphStatsRecorder = new RFPHStatsRecorder()

lazy val rfphRestrictStep = new RFPHRestrictStep()

lazy val rfphTakeStepUtil = new RFPHTakeStepUtil()(statsReceiver)

lazy val rfphPrerankFilter = new RFPHPrerankFilter()(statsReceiver)

lazy val rfphLightRanker = new RFPHLightRanker(lightRanker, statsReceiver)

lazy val sendHandlerPredicateUtil = new SendHandlerPredicateUtil()(statsReceiver)

lazy val ntabSender =

new NtabSender(

notificationServiceSender,

nTabHistoryStore,

notificationServiceDelete,

notificationServiceDeleteTimeline

)

lazy val ibis2Sender = new Ibis2Sender(pushIbisV2Store, tweetTranslationStore, statsReceiver)

lazy val historyWriter = new HistoryWriter(historyStore, statsReceiver)

lazy val loggedOutHistoryWriter = new HistoryWriter(loggedOutHistoryStore, statsReceiver)

lazy val eventBusWriter = new EventBusWriter(pushSendEventBusPublisher, statsReceiver)

lazy val ntabOnlyChannelSelector = new NtabOnlyChannelSelector

lazy val notificationSender =

new NotificationSender(

ibis2Sender,

ntabSender,

statsReceiver,

notificationScribe

)

lazy val candidateNotifier =

new CandidateNotifier(

notificationSender,

casLock = casLock,

historyWriter = historyWriter,

eventBusWriter = eventBusWriter,

ntabOnlyChannelSelector = ntabOnlyChannelSelector

)(statsReceiver)

lazy val loggedOutCandidateNotifier = new CandidateNotifier(

notificationSender,

casLock = casLock,

historyWriter = loggedOutHistoryWriter,

eventBusWriter = null,

ntabOnlyChannelSelector = ntabOnlyChannelSelector

)(statsReceiver)

lazy val rfphNotifier =

new RefreshForPushNotifier(rfphStatsRecorder, candidateNotifier)(statsReceiver)

lazy val loRfphNotifier =

new LoggedOutRefreshForPushNotifier(rfphStatsRecorder, loggedOutCandidateNotifier)(

statsReceiver)

lazy val rfphRanker = {

val randomRanker = RandomRanker[Target, PushCandidate]()

val subscriptionCreatorRanker =

new SubscriptionCreatorRanker(superFollowEligibilityUserStore, statsReceiver)

new RFPHRanker(

randomRanker,

weightedOpenOrNtabClickModelScorer,

subscriptionCreatorRanker,

userHealthSignalStore,

producerMediaRepresentationStore,

statsReceiver

)

}

lazy val rfphFeatureHydrator = new RFPHFeatureHydrator(featureHydrator)

lazy val loggedOutRFPHRanker = new LoggedOutRanker(cachedTweetyPieStoreV2, statsReceiver)

override val userFeaturesStore: ReadableStore[Long, UserFeatures] = {

implicit val valueCodec = new BinaryScalaCodec(UserFeatures)

val dataset: Dataset[Long, UserFeatures] =

Dataset(

"",

"user\_features\_pushservice\_apollo",

"recommendations\_user\_features\_apollo",

Apollo)

ObservedMemcachedReadableStore.fromCacheClient(

backingStore = ObservedReadableStore(buildManhattanStore(dataset))(

statsReceiver.scope("UncachedUserFeaturesStore")

),

cacheClient = pushServiceCacheClient,

ttl = 24.hours

)(

valueInjection = valueCodec,

statsReceiver = statsReceiver.scope("UserFeaturesStore"),

keyToString = { k: Long => s"ufts/$k" }

)

}

override def htlScoreStore(userId: Long): ReadableStore[Long, ScoredTweet] = {

val fetcher = new TimelineScorerTweetScoresV1ClientColumn(stratoClient).fetcher

val htlStore = buildStore(

StratoFetchableStore.withView[Long, TimelineScorerScoreView, ScoredTweet](

fetcher,

TimelineScorerScoreView(Some(userId))

),

"htlScoreStore"

)

htlStore

}

override val userTargetingPropertyStore: ReadableStore[Long, UserTargetingProperty] = {

val name = "userTargetingPropertyStore"

val store = StratoFetchableStore

.withUnitView(new TargetingPropertyOnUserClientColumn(stratoClient).fetcher)

buildStore(store, name)

}

override val timelinesUserSessionStore: ReadableStore[Long, UserSession] = {

implicit val valueCodec = new CompactScalaCodec(UserSession)

val dataset: Dataset[Long, UserSession] = Dataset[Long, UserSession](

"",

"frigate\_realgraph",

"real\_graph\_user\_features",

Apollo

)

ObservedMemcachedReadableStore.fromCacheClient(

backingStore = ObservedReadableStore(buildManhattanStore(dataset))(

statsReceiver.scope("UncachedTimelinesUserSessionStore")

),

cacheClient = pushServiceCacheClient,

ttl = 6.hours

)(

valueInjection = valueCodec,

statsReceiver = statsReceiver.scope("timelinesUserSessionStore"),

keyToString = { k: Long => s"tluss/$k" }

)

}

lazy val recentTweetsFromTflockStore: ReadableStore[Long, Seq[Long]] =

ObservedReadableStore(

RecentTweetsByAuthorsStore.usingRecentTweetsConfig(

tflockClient,

RecentTweetsConfig(maxResults = 1, maxAge = 3.days)

)

)(statsReceiver.scope("RecentTweetsFromTflockStore"))

lazy val recentTweetsByAuthorsStore: ReadableStore[RecentTweetsQuery, Seq[Seq[Long]]] =

ObservedReadableStore(

RecentTweetsByAuthorsStore(tflockClient)

)(statsReceiver.scope("RecentTweetsByAuthorsStore"))

val jobConfig = PopGeoInterestProvider

.getPopularTweetsJobConfig(

InterestDeployConfig(

AppId("PopularTweetsByInterestProd"),

Cluster.ATLA,

Env.Prod,

serviceIdentifier,

manhattanClientMtlsParams

))

.withManhattanAppId("frigate\_pop\_by\_geo\_tweets")

override val topTweetsByGeoStore = TopTweetsStore.withMemCache(

jobConfig,

pushServiceCacheClient,

10.seconds

)(statsReceiver)

override val topTweetsByGeoV2VersionedStore: ReadableStore[String, PopTweetsInPlace] = {

StratoFetchableStore.withUnitView[String, PopTweetsInPlace](

stratoClient,

"recommendations/popgeo/popGeoTweetsVersioned")

}

override lazy val pushcapDynamicPredictionStore: ReadableStore[Long, PushcapUserHistory] = {

StratoFetchableStore.withUnitView[Long, PushcapUserHistory](

stratoClient,

"frigate/magicrecs/pushcapDynamicPrediction.User")

}

override val tweetAuthorLocationFeatureBuilder =

UserLocationFeatureBuilder(Some("TweetAuthor"))

.withStats()

override val tweetAuthorLocationFeatureBuilderById =

UserLocationFeatureBuilderById(

userCountryStore,

tweetAuthorLocationFeatureBuilder

).withStats()

override val socialContextActionsFeatureBuilder =

SocialContextActionsFeatureBuilder().withStats()

override val tweetContentFeatureBuilder =

TweetContentFeatureBuilder(tweetContentFeatureCacheStore).withStats()

override val tweetAuthorRecentRealGraphFeatureBuilder =

RecentRealGraphFeatureBuilder(

stratoClient,

UserAuthorEntity,

TargetUserEntity,

TweetAuthorEntity,

TweetAuthorRecentRealGraphFeatures(statsReceiver.scope("TweetAuthorRecentRealGraphFeatures"))

).withStats()

override val socialContextRecentRealGraphFeatureBuilder =

SocialContextRecentRealGraphFeatureBuilder(

RecentRealGraphFeatureBuilder(

stratoClient,

TargetUserSocialContextEntity,

TargetUserEntity,

SocialContextEntity,

SocialContextRecentRealGraphFeatures(

statsReceiver.scope("SocialContextRecentRealGraphFeatures"))

)(statsReceiver

.scope("SocialContextRecentRealGraphFeatureBuilder").scope("RecentRealGraphFeatureBuilder"))

).withStats()

override val tweetSocialProofFeatureBuilder =

TweetSocialProofFeatureBuilder(Some("TargetUser")).withStats()

override val targetUserFullRealGraphFeatureBuilder =

TargetFullRealGraphFeatureBuilder(Some("TargetUser")).withStats()

override val postProcessingFeatureBuilder: PostProcessingFeatureBuilder =

PostProcessingFeatureBuilder()

override val mrOfflineUserCandidateSparseAggregatesFeatureBuilder =

MrOfflineUserCandidateSparseAggregatesFeatureBuilder(stratoClient, featureStoreUtil).withStats()

override val mrOfflineUserAggregatesFeatureBuilder =

MrOfflineUserAggregatesFeatureBuilder(stratoClient, featureStoreUtil).withStats()

override val mrOfflineUserCandidateAggregatesFeatureBuilder =

MrOfflineUserCandidateAggregatesFeatureBuilder(stratoClient, featureStoreUtil).withStats()

override val tweetAnnotationsFeatureBuilder =

TweetAnnotationsFeatureBuilder(stratoClient).withStats()

override val targetUserMediaRepresentationFeatureBuilder =

UserMediaRepresentationFeatureBuilder(userMediaRepresentationStore).withStats()

override val targetLevelFeatureBuilder =

TargetLevelFeatureBuilder(featureStoreUtil, targetLevelFeaturesConfig).withStats()

override val candidateLevelFeatureBuilder =

CandidateLevelFeatureBuilder(featureStoreUtil).withStats()

override lazy val targetFeatureHydrator = RelevanceTargetFeatureHydrator(

targetUserFullRealGraphFeatureBuilder,

postProcessingFeatureBuilder,

targetUserMediaRepresentationFeatureBuilder,

targetLevelFeatureBuilder

)

override lazy val featureHydrator =

FeatureHydrator(targetFeatureHydrator, candidateFeatureHydrator)

val pushServiceLightRankerConfig: LightRankerConfig = new LightRankerConfig(

pushserviceThriftClientId,

serviceIdentifier,

statsReceiver.scope("lightRanker"),

deepbirdv2PredictionServiceDest,

"DeepbirdV2PredictionService"

)

val lightRanker: MagicRecsServeDataRecordLightRanker =

pushServiceLightRankerConfig.lightRanker

override val tweetImpressionStore: ReadableStore[Long, Seq[Long]] = {

val name = "htl\_impression\_store"

val store = buildStore(

HtlTweetImpressionStore.createStoreWithTweetIds(

requestTimeout = 6.seconds,

label = "htl\_tweet\_impressions",

serviceIdentifier = serviceIdentifier,

statsReceiver = statsReceiver

),

name

)

val numTweetsReturned =

statsReceiver.scope(name).stat("num\_tweets\_returned\_per\_user")

new TransformedReadableStore(store)((userId: Long, tweetIds: Seq[Long]) => {

numTweetsReturned.add(tweetIds.size)

Future.value(Some(tweetIds))

})

}

val ruxTweetImpressionsStore = new TweetImpressionsStore(stratoClient)

override val strongTiesStore: ReadableStore[Long, STPResult] = {

implicit val valueCodec = new BinaryScalaCodec(STPResult)

val strongTieScoringDataset: Dataset[Long, STPResult] =

Dataset("", "frigate\_stp", "stp\_result\_rerank", Athena)

buildManhattanStore(strongTieScoringDataset)

}

override lazy val earlybirdFeatureStore = ObservedReadableStore(

EarlybirdFeatureStore(

clientId = pushserviceThriftClientId.name,

earlybirdSearchStore = earlybirdSearchStore

)

)(statsReceiver.scope("EarlybirdFeatureStore"))

override lazy val earlybirdFeatureBuilder = EarlybirdFeatureBuilder(earlybirdFeatureStore)

override lazy val earlybirdSearchStore = {

val earlybirdClientName: String = "earlybird"

val earlybirdSearchStoreName: String = "EarlybirdSearchStore"

val earlybirdClient = new EarlybirdService.FinagledClient(

readOnlyThriftService(

earlybirdClientName,

earlybirdSearchDest,

statsReceiver,

pushserviceThriftClientId,

tries = 1,

requestTimeout = 3.seconds,

mTLSServiceIdentifier = Some(serviceIdentifier)

),

clientParam = RichClientParam(protocolFactory = new TCompactProtocol.Factory)

)

ObservedReadableStore(

EarlybirdSearchStore(earlybirdClient)(statsReceiver.scope(earlybirdSearchStoreName))

)(statsReceiver.scope(earlybirdSearchStoreName))

}

override lazy val earlybirdCandidateSource: EarlybirdCandidateSource = EarlybirdCandidateSource(

clientId = pushserviceThriftClientId.name,

earlybirdSearchStore = earlybirdSearchStore

)

override val realGraphScoresTop500InStore: RealGraphScoresTop500InStore = {

val stratoRealGraphInStore =

StratoFetchableStore

.withUnitView[Long, CandidateSeq](

stratoClient,

"frigate/magicrecs/fanoutCoi500pRealGraphV2")

RealGraphScoresTop500InStore(

ObservedMemcachedReadableStore.fromCacheClient(

backingStore = stratoRealGraphInStore,

cacheClient = entityGraphCacheClient,

ttl = 24.hours

)(

valueInjection = BinaryScalaCodec(CandidateSeq),

statsReceiver = statsReceiver.scope("CachedRealGraphScoresTop500InStore"),

keyToString = { k: Long => s"500p\_test/$k" }

)

)

}

override val tweetEntityGraphStore = {

val tweetEntityGraphClient = new UserTweetEntityGraph.FinagledClient(

Finagle.readOnlyThriftService(

"user\_tweet\_entity\_graph",

userTweetEntityGraphDest,

statsReceiver,

pushserviceThriftClientId,

requestTimeout = 5.seconds,

mTLSServiceIdentifier = Some(serviceIdentifier)

)

)

ObservedReadableStore(

RecommendedTweetEntitiesStore(

tweetEntityGraphClient,

statsReceiver.scope("RecommendedTweetEntitiesStore")

)

)(statsReceiver.scope("RecommendedTweetEntitiesStore"))

}

override val userUserGraphStore = {

val userUserGraphClient = new UserUserGraph.FinagledClient(

Finagle.readOnlyThriftService(

"user\_user\_graph",

userUserGraphDest,

statsReceiver,

pushserviceThriftClientId,

requestTimeout = 5.seconds,

mTLSServiceIdentifier = Some(serviceIdentifier)

),

clientParam = RichClientParam(serviceName = "user\_user\_graph")

)

ObservedReadableStore(

UserUserGraphStore(userUserGraphClient, statsReceiver.scope("UserUserGraphStore"))

)(statsReceiver.scope("UserUserGraphStore"))

}

override val ntabCaretFeedbackStore: ReadableStore[GenericNotificationsFeedbackRequest, Seq[

CaretFeedbackDetails

]] = {

val client = ManhattanKVClient(

"pushservice\_ntab\_caret\_feedback\_omega",

Omega.wilyName,

manhattanClientMtlsParams

)

val endpoint = ManhattanKVEndpointBuilder(client)

.defaultGuarantee(Guarantee.SoftDcReadMyWrites)

.defaultMaxTimeout(3.seconds)

.maxRetryCount(2)

.statsReceiver(statsReceiver)

.build()

val feedbackSignalManhattanClient =

FeedbackSignalManhattanClient(endpoint, statsReceiver.scope("FeedbackSignalManhattanClient"))

NtabCaretFeedbackStore(feedbackSignalManhattanClient)

}

override val genericFeedbackStore: ReadableStore[FeedbackRequest, Seq[

FeedbackPromptValue

]] = {

FeedbackStore(

GenericFeedbackStoreBuilder.build(

manhattanKVClientAppId = "frigate\_pushservice\_ntabfeedback\_prompt",

environment = NotifEnvironment.apply(serviceIdentifier.environment),

svcIdentifier = serviceIdentifier,

statsReceiver = statsReceiver

))

}

override val genericNotificationFeedbackStore: GenericFeedbackStore = {

GenericFeedbackStoreBuilder.build(

manhattanKVClientAppId = "frigate\_pushservice\_ntabfeedback\_prompt",

environment = NotifEnvironment.apply(serviceIdentifier.environment),

svcIdentifier = serviceIdentifier,

statsReceiver = statsReceiver

)

}

override val earlybirdSearchDest = "/s/earlybird-root-superroot/root-superroot"

// low latency as compared to default `semanticCoreMetadataClient`

private val lowLatencySemanticCoreMetadataClient: MetadataService.MethodPerEndpoint =

new MetadataService.FinagledClient(

Finagle.readOnlyThriftService(

name = "semantic\_core\_metadata\_service",

dest = "/s/escherbird/metadataservice",

statsReceiver = statsReceiver,

thriftClientId = pushserviceThriftClientId,

tries = 2, // total number of tries. number of retries = tries - 1

requestTimeout = 2.seconds,

mTLSServiceIdentifier = Some(serviceIdentifier)

)

)

private val semanticCoreMetadataStitchClient = new MetadataStitchClient(

lowLatencySemanticCoreMetadataClient

)

override val semanticCoreMegadataStore: ReadableStore[SemanticEntityForQuery, EntityMegadata] = {

val name = "semantic\_core\_megadata\_store\_cached"

val store = MetaDataReadableStore.getMegadataReadableStore(

metadataStitchClient = semanticCoreMetadataStitchClient,

typedMetadataDomains = Some(Set(Domains.EventsEntityService))

)

ObservedCachedReadableStore

.from(

store = ObservedReadableStore(store)(

statsReceiver

.scope("store")

.scope("semantic\_core\_megadata\_store")

),

ttl = 1.hour,

maxKeys = 1000,

cacheName = "semantic\_core\_megadata\_cache",

windowSize = 10000L

)(statsReceiver.scope("store", name))

}

override val basketballGameScoreStore: ReadableStore[QualifiedId, BasketballGameLiveUpdate] = {

StratoFetchableStore.withUnitView[QualifiedId, BasketballGameLiveUpdate](

stratoClient,

"semanticCore/basketballGameScore.Entity")

}

override val baseballGameScoreStore: ReadableStore[QualifiedId, BaseballGameLiveUpdate] = {

StratoFetchableStore.withUnitView[QualifiedId, BaseballGameLiveUpdate](

stratoClient,

"semanticCore/baseballGameScore.Entity")

}

override val cricketMatchScoreStore: ReadableStore[QualifiedId, CricketMatchLiveUpdate] = {

StratoFetchableStore.withUnitView[QualifiedId, CricketMatchLiveUpdate](

stratoClient,

"semanticCore/cricketMatchScore.Entity")

}

override val soccerMatchScoreStore: ReadableStore[QualifiedId, SoccerMatchLiveUpdate] = {

ObservedCachedReadableStore

.from(

store = StratoFetchableStore.withUnitView[QualifiedId, SoccerMatchLiveUpdate](

stratoClient,

"semanticCore/soccerMatchScore.Entity"),

ttl = 10.seconds,

maxKeys = 100,

cacheName = "SoccerMatchCachedStore",

windowSize = 100L

)(statsReceiver.scope("SoccerMatchCachedStore"))

}

override val nflGameScoreStore: ReadableStore[QualifiedId, NflFootballGameLiveUpdate] = {

ObservedCachedReadableStore

.from(

store = StratoFetchableStore.withUnitView[QualifiedId, NflFootballGameLiveUpdate](

stratoClient,

"semanticCore/nflFootballGameScore.Entity"),

ttl = 10.seconds,

maxKeys = 100,

cacheName = "NFLMatchCachedStore",

windowSize = 100L

)(statsReceiver.scope("NFLMatchCachedStore"))

}

override val userHealthSignalStore: ReadableStore[Long, UserHealthSignalResponse] = {

val userHealthSignalFetcher =

stratoClient.fetcher[Long, Seq[UserHealthSignal], UserHealthSignalResponse](

"hss/user\_signals/api/healthSignals.User"

)

val store = buildStore(

StratoFetchableStore.withView[Long, Seq[UserHealthSignal], UserHealthSignalResponse](

userHealthSignalFetcher,

Seq(

AgathaRecentAbuseStrikeDouble,

AgathaCalibratedNsfwDouble,

AgathaCseDouble,

NsfwTextUserScoreDouble,

NsfwConsumerScoreDouble)),

"UserHealthSignalFetcher"

)

if (!inMemCacheOff) {

ObservedCachedReadableStore

.from(

store = ObservedReadableStore(store)(

statsReceiver.scope("store").scope("user\_health\_model\_score\_store")),

ttl = 12.hours,

maxKeys = 16777215,

cacheName = "user\_health\_model\_score\_store\_cache",

windowSize = 10000L

)(statsReceiver.scope("store", "user\_health\_model\_score\_store\_cached"))

} else {

store

}

}

override val tweetHealthScoreStore: ReadableStore[TweetScoringRequest, TweetScoringResponse] = {

val tweetHealthScoreFetcher =

stratoClient.fetcher[TweetScoringRequest, Unit, TweetScoringResponse](

"abuse/detection/tweetHealthModelScore"

)

val store = buildStore(

StratoFetchableStore.withUnitView(tweetHealthScoreFetcher),

"TweetHealthScoreFetcher"

)

ObservedCachedReadableStore

.from(

store = ObservedReadableStore(store)(

statsReceiver.scope("store").scope("tweet\_health\_model\_score\_store")),

ttl = 30.minutes,

maxKeys = 1000,

cacheName = "tweet\_health\_model\_score\_store\_cache",

windowSize = 10000L

)(statsReceiver.scope("store", "tweet\_health\_model\_score\_store\_cached"))

}

override val appPermissionStore: ReadableStore[(Long, (String, String)), AppPermission] = {

val store = StratoFetchableStore

.withUnitView[(Long, (String, String)), AppPermission](

stratoClient,

"clients/permissionsState")

ObservedCachedReadableStore.from[(Long, (String, String)), AppPermission](

buildStore(store, "mr\_app\_permission\_store"),

ttl = 30.minutes,

maxKeys = 1000,

cacheName = "mr\_app\_permission\_store\_cache",

windowSize = 10000L

)(statsReceiver.scope("mr\_app\_permission\_store\_cached"))

}

def pushSendEventStreamName: String

override val pushSendEventBusPublisher = EventBusPublisherBuilder()

.clientId("frigate\_pushservice")

.streamName(pushSendEventStreamName)

.thriftStruct(NotificationScribe)

.statsReceiver(statsReceiver.scope("push\_send\_eventbus"))

.build()

override lazy val candidateFeatureHydrator: CandidateFeatureHydrator =

CandidateFeatureHydrator(

socialContextActionsFeatureBuilder = Some(socialContextActionsFeatureBuilder),

tweetSocialProofFeatureBuilder = Some(tweetSocialProofFeatureBuilder),

earlybirdFeatureBuilder = Some(earlybirdFeatureBuilder),

tweetContentFeatureBuilder = Some(tweetContentFeatureBuilder),

tweetAuthorRecentRealGraphFeatureBuilder = Some(tweetAuthorRecentRealGraphFeatureBuilder),

socialContextRecentRealGraphFeatureBuilder = Some(socialContextRecentRealGraphFeatureBuilder),

tweetAnnotationsFeatureBuilder = Some(tweetAnnotationsFeatureBuilder),

mrOfflineUserCandidateSparseAggregatesFeatureBuilder =

Some(mrOfflineUserCandidateSparseAggregatesFeatureBuilder),

candidateLevelFeatureBuilder = Some(candidateLevelFeatureBuilder)

)(statsReceiver.scope("push\_feature\_hydrator"))

private val candidateCopyCross =

new CandidateCopyExpansion(statsReceiver.scope("refresh\_handler/cross"))

override lazy val candidateHydrator: PushCandidateHydrator =

PushCandidateHydrator(

this.socialGraphServiceProcessStore,

safeUserStore,

listAPIStore,

candidateCopyCross)(

statsReceiver.scope("push\_candidate\_hydrator"),

weightedOpenOrNtabClickModelScorer)

override lazy val sendHandlerCandidateHydrator: SendHandlerPushCandidateHydrator =

SendHandlerPushCandidateHydrator(

lexServiceStore,

fanoutMetadataStore,

semanticCoreMegadataStore,

safeUserStore,

simClusterToEntityStore,

audioSpaceStore,

interestsWithLookupContextStore,

uttEntityHydrationStore,

superFollowCreatorTweetCountStore

)(

statsReceiver.scope("push\_candidate\_hydrator"),

weightedOpenOrNtabClickModelScorer

)

def mrRequestScriberNode: String

def loggedOutMrRequestScriberNode: String

override lazy val configParamsBuilder: ConfigParamsBuilder = ConfigParamsBuilder(

config = overridesConfig,

featureContextBuilder = FeatureContextBuilder(featureSwitches),

statsReceiver = statsReceiver

)

def buildStore[K, V](store: ReadableStore[K, V], name: String): ReadableStore[K, V] = {

ObservedReadableStore(store)(statsReceiver.scope("store").scope(name))

}

def buildManhattanStore[K, V](dataset: Dataset[K, V]): ReadableStore[K, V] = {

val manhattanKVClientParams = ManhattanKVClientMtlsParams(

serviceIdentifier = serviceIdentifier,

opportunisticTls = OpportunisticTls.Required

)

ManhattanStore

.fromDatasetWithMtls[K, V](

dataset,

mtlsParams = manhattanKVClientParams,

statsReceiver = statsReceiver.scope(dataset.datasetName))

}

def buildCachedTweetyPieStore(

getTweetOptions: Option[GetTweetOptions],

keyPrefix: String

): ReadableStore[Long, TweetyPieResult] = {

def discardAdditionalMediaInfo(tweetypieResult: TweetyPieResult) = {

val updatedMedia = tweetypieResult.tweet.media.map { mediaSeq =>

mediaSeq.map { media => media.copy(additionalMetadata = None, sizes = Nil.toSet) }

}

val updatedTweet = tweetypieResult.tweet.copy(media = updatedMedia)

tweetypieResult.copy(tweet = updatedTweet)

}

val tweetypieStoreWithoutAdditionalMediaInfo = TweetyPieStore(

tweetyPieClient,

getTweetOptions,

transformTweetypieResult = discardAdditionalMediaInfo

)(statsReceiver.scope("tweetypie\_without\_additional\_media\_info"))

ObservedMemcachedReadableStore.fromCacheClient(

backingStore = tweetypieStoreWithoutAdditionalMediaInfo,

cacheClient = pushServiceCoreSvcsCacheClient,

ttl = 12.hours

)(

valueInjection = TweetyPieResultInjection,

statsReceiver = statsReceiver.scope("TweetyPieStore"),

keyToString = { k: Long => s"$keyPrefix/$k" }

)

}

override def init(): Future[Unit] =

ClientRegistry.expAllRegisteredClientsResolved().map { clients =>

log.info("Done resolving clients: " + clients.mkString("[", ", ", "]"))

}

val InlineActionsMhColumn =

"frigate/magicrecs/inlineActionsMh"

override val inlineActionHistoryStore: ReadableStore[Long, Seq[(Long, String)]] =

StratoScannableStore

.withUnitView[(Long, Slice[Long]), (Long, Long), String](stratoClient, InlineActionsMhColumn)

.composeKeyMapping[Long] { userId =>

(userId, Slice[Long](from = None, to = None, limit = None))

}.mapValues { response =>

response.map {

case (key, value) => (key.\_2, value)

}

}

override val tripTweetCandidateStore: ReadableStore[TripDomain, TripTweets] = {

StratoFetchableStore

.withUnitView[TripDomain, TripTweets](

new TripTweetsAirflowProdClientColumn(stratoClient).fetcher)

}

override val softUserFollowingStore: ReadableStore[User, Seq[Long]] = new SoftUserFollowingStore(

stratoClient)

override val superFollowEligibilityUserStore: ReadableStore[Long, Boolean] = {

StratoFetchableStore.withUnitView[Long, Boolean](

stratoClient,

"audiencerewards/audienceRewardsService/getSuperFollowEligibility.User")

}

override val superFollowCreatorTweetCountStore: ReadableStore[UserId, Int] = {

ObservedCachedReadableStore

.from(

store = StratoFetchableStore

.withUnitView[UserId, Int](new CreatorSubscriptionNumTweetsColumn(stratoClient).fetcher),

ttl = 5.minutes,

maxKeys = 1000,

cacheName = "SuperFollowCreatorTweetCountStore",

windowSize = 10000L

)(statsReceiver.scope("SuperFollowCreatorTweetCountStore"))

}

override val hasSuperFollowingRelationshipStore: ReadableStore[

HasSuperFollowingRelationshipRequest,

Boolean

] = {

StratoFetchableStore.withUnitView[HasSuperFollowingRelationshipRequest, Boolean](

stratoClient,

"audiencerewards/superFollows/hasSuperFollowingRelationshipV2")

}

override val superFollowApplicationStatusStore: ReadableStore[

(Long, SellerTrack),

SellerApplicationState

] = {

StratoFetchableStore.withUnitView[(Long, SellerTrack), SellerApplicationState](

stratoClient,

"periscope/eligibility/applicationStatus")

}

def historyStoreMemcacheDest: String

override lazy val recentHistoryCacheClient = {

RecentHistoryCacheClient.build(historyStoreMemcacheDest, serviceIdentifier, statsReceiver)

}

override val openAppUserStore: ReadableStore[Long, Boolean] = {

buildStore(OpenAppUserStore(stratoClient), "OpenAppUserStore")

}

}