package com.twitter.tweetypie.config

import com.twitter.decider.Decider

import com.twitter.decider.DeciderFactory

import com.twitter.decider.LocalOverrides

import com.twitter.featureswitches.v2.builder.FeatureSwitchesBuilder

import com.twitter.finagle.filter.DarkTrafficFilter

import com.twitter.finagle.stats.DefaultStatsReceiver

import com.twitter.finagle.stats.NullStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.thrift.Protocols

import com.twitter.finagle.util.DefaultTimer

import com.twitter.finagle.Filter

import com.twitter.finagle.Service

import com.twitter.finagle.SimpleFilter

import com.twitter.quill.capture.\_

import com.twitter.servo.util.MemoizingStatsReceiver

import com.twitter.servo.util.WaitForServerSets

import com.twitter.tweetypie.ThriftTweetService

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

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

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

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

import com.twitter.tweetypie.context.TweetypieContext

import com.twitter.tweetypie.matching.Tokenizer

import com.twitter.tweetypie.service.\_

import com.twitter.tweetypie.thriftscala.TweetServiceInternal$FinagleService

import com.twitter.util.\_

import com.twitter.util.logging.Logger

import scala.util.control.NonFatal

class TweetServerBuilder(settings: TweetServiceSettings) {

/\*\*

\* A logger used by some of the built-in initializers.

\*/

val log: Logger = Logger(getClass)

/\*\*

\* The top-level stats receiver. Defaults to the default StatsReceiver

\* embedded in Finagle.

\*/

val statsReceiver: StatsReceiver =

new MemoizingStatsReceiver(DefaultStatsReceiver)

val hostStatsReceiver: StatsReceiver =

if (settings.clientHostStats)

statsReceiver

else

NullStatsReceiver

/\*\*

\* A timer for scheduling various things.

\*/

val timer: Timer = DefaultTimer

/\*\*

\* Creates a decider instance by looking up the decider configuration information

\* from the settings object.

\*/

val decider: Decider = {

val fileBased = DeciderFactory(settings.deciderBaseFilename, settings.deciderOverlayFilename)()

// Use the tweetypie decider dashboard name for propagating decider overrides.

LocalOverrides.decider("tweetypie").orElse(fileBased)

}

val deciderGates: TweetypieDeciderGates = {

val deciderGates = TweetypieDeciderGates(decider, settings.deciderOverrides)

// Write out the configuration overrides to the log so that it's

// easy to confirm how this instance has been customized.

deciderGates.overrides.foreach {

case (overrideName, overrideValue) =>

log.info("Decider feature " + overrideName + " overridden to " + overrideValue)

if (deciderGates.unusedOverrides.contains(overrideName)) {

log.error("Unused decider override flag: " + overrideName)

}

}

val scopedReceiver = statsReceiver.scope("decider\_values")

deciderGates.availabilityMap.foreach {

case (feature, value) =>

scopedReceiver.provideGauge(feature) {

// Default value of -1 indicates error state.

value.getOrElse(-1).toFloat

}

}

deciderGates

}

val featureSwitchesWithExperiments = FeatureSwitchesBuilder

.createWithExperiments("/features/tweetypie/main")

.build()

val featureSwitchesWithoutExperiments = FeatureSwitchesBuilder

.createWithNoExperiments("/features/tweetypie/main", Some(statsReceiver))

.build()

// \*\*\*\*\*\*\*\*\* initializer \*\*\*\*\*\*\*\*\*\*

private[this] def warmupTextTokenization(logger: Logger): Unit = {

logger.info("Warming up text tokenization")

val watch = Stopwatch.start()

Tokenizer.warmUp()

logger.info(s"Warmed up text tokenization in ${watch()}")

}

private[this] def runWarmup(tweetService: Activity[ThriftTweetService]): Unit = {

val tokenizationLogger = Logger("com.twitter.tweetypie.TweetServerBuilder.TokenizationWarmup")

warmupTextTokenization(tokenizationLogger)

val warmupLogger = Logger("com.twitter.tweetypie.TweetServerBuilder.BackendWarmup")

// #1 warmup backends

Await.ready(settings.backendWarmupSettings(backendClients, warmupLogger, timer))

// #2 warmup Tweet Service

Await.ready {

tweetService.values.toFuture.map(\_.get).map { service =>

settings.warmupRequestsSettings.foreach(new TweetServiceWarmer(\_)(service))

}

}

}

private[this] def waitForServerSets(): Unit = {

val names = backendClients.referencedNames

val startTime = Time.now

log.info("will wait for serversets: " + names.mkString("\n", "\t\n", ""))

try {

Await.result(WaitForServerSets.ready(names, settings.waitForServerSetsTimeout, timer))

val duration = Time.now.since(startTime)

log.info("resolved all serversets in " + duration)

} catch {

case NonFatal(ex) => log.warn("failed to resolve all serversets", ex)

}

}

private[this] def initialize(tweetService: Activity[ThriftTweetService]): Unit = {

waitForServerSets()

runWarmup(tweetService)

// try to force a GC before starting to serve requests; this may or may not do anything

System.gc()

}

// \*\*\*\*\*\*\*\*\* builders \*\*\*\*\*\*\*\*\*\*

val clientIdHelper = new ClientIdHelper(

new ConditionalServiceIdentifierStrategy(

condition = deciderGates.preferForwardedServiceIdentifierForClientId,

ifTrue = PreferForwardedServiceIdentifierForStrato,

ifFalse = UseTransportServiceIdentifier,

),

)

val backendClients: BackendClients =

BackendClients(

settings = settings,

deciderGates = deciderGates,

statsReceiver = statsReceiver,

hostStatsReceiver = hostStatsReceiver,

timer = timer,

clientIdHelper = clientIdHelper,

)

val tweetService: Activity[ThriftTweetService] =

TweetServiceBuilder(

settings = settings,

statsReceiver = statsReceiver,

timer = timer,

deciderGates = deciderGates,

featureSwitchesWithExperiments = featureSwitchesWithExperiments,

featureSwitchesWithoutExperiments = featureSwitchesWithoutExperiments,

backendClients = backendClients,

clientIdHelper = clientIdHelper,

)

// Strato columns should use this tweetService

def stratoTweetService: Activity[ThriftTweetService] =

tweetService.map { service =>

// Add quill functionality to the strato tweet service only

val quillCapture = QuillCaptureBuilder(settings, deciderGates)

new QuillTweetService(quillCapture, service)

}

def build: Activity[Service[Array[Byte], Array[Byte]]] = {

val quillCapture = QuillCaptureBuilder(settings, deciderGates)

val darkTrafficFilter: SimpleFilter[Array[Byte], Array[Byte]] =

if (!settings.trafficForkingEnabled) {

Filter.identity

} else {

new DarkTrafficFilter(

backendClients.darkTrafficClient,

\_ => deciderGates.forkDarkTraffic(),

statsReceiver

)

}

val serviceFilter =

quillCapture

.getServerFilter(ThriftProto.server)

.andThen(TweetypieContext.Local.filter[Array[Byte], Array[Byte]])

.andThen(darkTrafficFilter)

initialize(tweetService)

// tweetService is an Activity[ThriftTweetService], so this callback

// is called every time that Activity updates (on ConfigBus changes).

tweetService.map { service =>

val finagleService =

new TweetServiceInternal$FinagleService(

service,

protocolFactory = Protocols.binaryFactory(),

stats = NullStatsReceiver,

maxThriftBufferSize = settings.maxThriftBufferSize

)

serviceFilter andThen finagleService

}

}

}

object QuillCaptureBuilder {

val tweetServiceWriteMethods: Set[String] =

Set(

"async\_delete",

"async\_delete\_additional\_fields",

"async\_erase\_user\_tweets",

"async\_incr\_fav\_count",

"async\_insert",

"async\_set\_additional\_fields",

"async\_set\_retweet\_visibility",

"async\_takedown",

"async\_undelete\_tweet",

"async\_update\_possibly\_sensitive\_tweet",

"cascaded\_delete\_tweet",

"delete\_additional\_fields",

"delete\_retweets",

"delete\_tweets",

"erase\_user\_tweets",

"flush",

"incr\_fav\_count",

"insert",

"post\_retweet",

"post\_tweet",

"remove",

"replicated\_delete\_additional\_fields",

"replicated\_delete\_tweet",

"replicated\_delete\_tweet2",

"replicated\_incr\_fav\_count",

"replicated\_insert\_tweet2",

"replicated\_scrub\_geo",

"replicated\_set\_additional\_fields",

"replicated\_set\_has\_safety\_labels",

"replicated\_set\_retweet\_visibility",

"replicated\_takedown",

"replicated\_undelete\_tweet2",

"replicated\_update\_possibly\_sensitive\_tweet",

"scrub\_geo",

"scrub\_geo\_update\_user\_timestamp",

"set\_additional\_fields",

"set\_has\_safety\_labels",

"set\_retweet\_visibility",

"set\_tweet\_user\_takedown",

"takedown",

"undelete\_tweet"

)

val tweetServiceReadMethods: Set[String] =

Set(

"get\_tweet\_counts",

"get\_tweet\_fields",

"get\_tweets",

"replicated\_get\_tweet\_counts",

"replicated\_get\_tweet\_fields",

"replicated\_get\_tweets"

)

def apply(settings: TweetServiceSettings, deciderGates: TweetypieDeciderGates): QuillCapture = {

val writesStore = SimpleScribeMessageStore("tweetypie\_writes")

.enabledBy(deciderGates.logWrites)

val readsStore = SimpleScribeMessageStore("tweetypie\_reads")

.enabledBy(deciderGates.logReads)

val messageStore =

MessageStore.selected {

case msg if tweetServiceWriteMethods.contains(msg.name) => writesStore

case msg if tweetServiceReadMethods.contains(msg.name) => readsStore

case \_ => writesStore

}

new QuillCapture(Store.legacyStore(messageStore), Some(settings.thriftClientId.name))

}

}