package com.twitter.tweetypie

package config

import com.twitter.app.Flag

import com.twitter.app.Flaggable

import com.twitter.app.Flags

import com.twitter.finagle.http.HttpMuxer

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

import com.twitter.finagle.mtls.authorization.server.MtlsServerSessionTrackerFilter

import com.twitter.finagle.mtls.server.MtlsStackServer.\_

import com.twitter.finagle.param.Reporter

import com.twitter.finagle.ssl.OpportunisticTls

import com.twitter.finagle.util.NullReporterFactory

import com.twitter.finagle.Thrift

import com.twitter.finagle.ThriftMux

import com.twitter.flockdb.client.thriftscala.Priority

import com.twitter.inject.Injector

import com.twitter.inject.annotations.{Flags => InjectFlags}

import com.twitter.scrooge.ThriftEnum

import com.twitter.scrooge.ThriftEnumObject

import com.twitter.server.handler.IndexHandler

import com.twitter.strato.catalog.Catalog

import com.twitter.strato.fed.StratoFed

import com.twitter.strato.fed.server.StratoFedServer

import com.twitter.strato.util.Ref

import com.twitter.strato.warmup.Warmer

import com.twitter.tweetypie.federated.StratoCatalogBuilder

import com.twitter.tweetypie.federated.warmups.StratoCatalogWarmups

import com.twitter.tweetypie.serverutil.ActivityService

import java.net.InetSocketAddress

import scala.reflect.ClassTag

object Env extends Enumeration {

val dev: Env.Value = Value

val staging: Env.Value = Value

val prod: Env.Value = Value

}

class TweetServiceFlags(flag: Flags, injector: => Injector) {

implicit object EnvFlaggable extends Flaggable[Env.Value] {

def parse(s: String): Env.Value =

s match {

// Handle Aurora env names that are different from tweetypie's names

case "devel" => Env.dev

case "test" => Env.staging

// Handle Tweetypie env names

case other => Env.withName(other)

}

}

val zone: Flag[String] =

flag("zone", "localhost", "One of: atla, pdxa, localhost, etc.")

val env: Flag[Env.Value] =

flag("env", Env.dev, "One of: testbox, dev, staging, prod")

val twemcacheDest: Flag[String] =

flag(

"twemcacheDest",

"/s/cache/tweetypie:twemcaches",

"The Name for the tweetypie cache cluster."

)

val deciderOverrides: Flag[Map[String, Boolean]] =

flag(

"deciderOverrides",

Map.empty[String, Boolean],

"Set deciders to constant values, overriding decider configuration files."

)(

// Unfortunately, the implicit Flaggable[Boolean] has a default

// value and Flaggable.ofMap[K, V] requires that the implicit

// Flaggable[V] not have a default. Even less fortunately, it

// doesn't say why. We're stuck with this.

Flaggable.ofMap(implicitly, Flaggable.mandatory(\_.toBoolean))

)

// "/decider.yml" comes from the resources included at

// "tweetypie/server/config", so you should not normally need to

// override this value. This flag is defined as a step toward making

// our command-line usage more similar to the standard

// twitter-server-internal flags.

def deciderBase(): String =

injector.instance[String](InjectFlags.named("decider.base"))

// Omitting a value for decider overlay flag causes the server to use

// only the static decider.

def deciderOverlay(): String =

injector.instance[String](InjectFlags.named("decider.overlay"))

// Omitting a value for the VF decider overlay flag causes the server

// to use only the static decider.

val vfDeciderOverlay: Flag[String] =

flag(

"vf.decider.overlay",

"The location of the overlay decider configuration for Visibility Filtering")

/\*\*

\* Warmup Requests happen as part of the initialization process, before any real requests are

\* processed. This prevents real requests from ever being served from a competely cold state

\*/

val enableWarmupRequests: Flag[Boolean] =

flag(

"enableWarmupRequests",

true,

"""| warms up Tweetypie service by generating random requests

| to Tweetypie that are processed prior to the actual client requests """.stripMargin

)

val grayListRateLimit: Flag[Double] =

flag("graylistRateLimit", 5.0, "rate-limit for non-allowlisted clients")

val servicePort: Flag[InetSocketAddress] =

flag("service.port", "port for tweet-service thrift interface")

val clientId: Flag[String] =

flag("clientId", "tweetypie.staging", "clientId to send in requests")

val allowlist: Flag[Boolean] =

flag("allowlist", true, "enforce client allowlist")

val clientHostStats: Flag[Boolean] =

flag("clientHostStats", false, "enable per client host stats")

val withCache: Flag[Boolean] =

flag("withCache", true, "if set to false, Tweetypie will launch without memcache")

/\*\*

\* Make any [[ThriftEnum]] value parseable as a [[Flag]] value. This

\* will parse case-insensitive values that match the unqualified

\* names of the values of the enumeration, in the manner of

\* [[ThriftEnum]]'s `valueOf` method.

\*

\* Consider a [[ThriftEnum]] generated from the following Thrift IDL snippet:

\*

\* {{{

\* enum Priority {

\* Low = 1

\* Throttled = 2

\* High = 3

\* }

\* }}}

\*

\* To enable defining flags that specify one of these enum values:

\*

\* {{{

\* implicit val flaggablePriority: Flaggable[Priority] = flaggableThriftEnum(Priority)

\* }}}

\*

\* In this example, the enumeration value `Priority.Low` can be

\* represented as the string "Low", "low", or "LOW".

\*/

def flaggableThriftEnum[T <: ThriftEnum: ClassTag](enum: ThriftEnumObject[T]): Flaggable[T] =

Flaggable.mandatory[T] { stringValue: String =>

enum

.valueOf(stringValue)

.getOrElse {

val validValues = enum.list.map(\_.name).mkString(", ")

throw new IllegalArgumentException(

s"Invalid value ${stringValue}. Valid values include: ${validValues}"

)

}

}

implicit val flaggablePriority: Flaggable[Priority] = flaggableThriftEnum(Priority)

val backgroundIndexingPriority: Flag[Priority] =

flag(

"backgroundIndexingPriority",

Priority.Low,

"specifies the queue to use for \"background\" tflock operations, such as removing edges " +

"for deleted Tweets. This exists for testing scenarios, when it is useful to see the " +

"effects of background indexing operations sooner. In production, this should always be " +

"set to \"low\" (the default)."

)

val tflockPageSize: Flag[Int] =

flag("tflockPageSize", 1000, "Number of items to return in each page when querying tflock")

val enableInProcessCache: Flag[Boolean] =

flag(

"enableInProcessCache",

true,

"if set to false, Tweetypie will not use the in-process cache"

)

val inProcessCacheSize: Flag[Int] =

flag("inProcessCacheSize", 1700, "maximum items in in-process cache")

val inProcessCacheTtlMs: Flag[Int] =

flag("inProcessCacheTtlMs", 10000, "milliseconds that hot keys are stored in memory")

val memcachePendingRequestLimit: Flag[Int] =

flag(

"memcachePendingRequestLimit",

100,

"Number of requests that can be queued on a single memcache connection (4 per cache server)"

)

val instanceId: Flag[Int] =

flag(

"configbus.instanceId",

-1,

"InstanceId of the tweetypie service instance for staged configuration distribution"

)

val instanceCount: Flag[Int] =

flag(

"configbus.instanceCount",

-1,

"Total number of tweetypie service instances for staged configuration distribution"

)

def serviceIdentifier(): ServiceIdentifier =

injector.instance[ServiceIdentifier]

val enableReplication: Flag[Boolean] =

flag(

"enableReplication",

true,

"Enable replication of reads (configurable via tweetypie\_replicate\_reads decider) and writes (100%) via DRPC"

)

val simulateDeferredrpcCallbacks: Flag[Boolean] =

flag(

"simulateDeferredrpcCallbacks",

false,

"""|For async write path, call back into current instance instead of via DRPC.

|This is used for test and devel instances so we can ensure the test traffic

|is going to the test instance.""".stripMargin

)

val shortCircuitLikelyPartialTweetReadsMs: Flag[Int] =

flag(

"shortCircuitLikelyPartialTweetReadsMs",

1500,

"""|Specifies a number of milliseconds before which we will short-circuit likely

|partial reads from MH and return a NotFound tweet response state. After

|experimenting we went with 1500 ms.""".stripMargin

)

val stringCenterProjects: Flag[Seq[String]] =

flag(

"stringcenter.projects",

Seq.empty[String],

"String Center project names, comma separated")(Flaggable.ofSeq(Flaggable.ofString))

val languagesConfig: Flag[String] =

flag("international.languages", "Supported languages config file")

}

class TweetypieMain extends StratoFedServer {

override def dest: String = "/s/tweetypie/tweetypie:federated"

val tweetServiceFlags: TweetServiceFlags = new TweetServiceFlags(flag, injector)

// display all the registered HttpMuxer handlers

HttpMuxer.addHandler("", new IndexHandler)

private[this] lazy val serverBuilder = {

val settings = new TweetServiceSettings(tweetServiceFlags)

val serverBuilder = new TweetServerBuilder(settings)

val mtlsSessionTrackerFilter =

new MtlsServerSessionTrackerFilter[Array[Byte], Array[Byte]](statsReceiver)

val mtlsTrackedService = mtlsSessionTrackerFilter.andThen(ActivityService(serverBuilder.build))

val thriftMuxServer = ThriftMux.server

// by default, finagle logs exceptions to chickadee, which is deprecated and

// basically unused. to avoid wasted overhead, we explicitly disable the reporter.

.configured(Reporter(NullReporterFactory))

.withLabel("tweetypie")

.withMutualTls(tweetServiceFlags.serviceIdentifier())

.withOpportunisticTls(OpportunisticTls.Required)

.configured(Thrift.param.ServiceClass(Some(classOf[ThriftTweetService])))

.serve(tweetServiceFlags.servicePort(), mtlsTrackedService)

closeOnExit(thriftMuxServer)

await(thriftMuxServer)

serverBuilder

}

override def configureRefCatalog(

catalog: Ref[Catalog[StratoFed.Column]]

): Ref[Catalog[StratoFed.Column]] =

catalog

.join {

Ref(

serverBuilder.stratoTweetService.flatMap { tweetService =>

StratoCatalogBuilder.catalog(

tweetService,

serverBuilder.backendClients.stratoserverClient,

serverBuilder.backendClients.gizmoduck.getById,

serverBuilder.backendClients.callbackPromotedContentLogger,

statsReceiver,

serverBuilder.deciderGates.enableCommunityTweetCreates,

)

}

)

}

.map { case (l, r) => l ++ r }

override def configureWarmer(warmer: Warmer): Unit = {

new TweetServiceSettings(tweetServiceFlags).warmupRequestsSettings.foreach { warmupSettings =>

warmer.add(

"tweetypie strato catalog",

() => StratoCatalogWarmups.warmup(warmupSettings, composedOps)

)

}

}

}

object Main extends TweetypieMain