package com.twitter.visibility.features

import com.twitter.finagle.mux.ClientDiscardedRequestException

import com.twitter.finagle.stats.NullStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.stitch.Stitch

import scala.language.existentials

class MissingFeatureException(feature: Feature[\_]) extends Exception("Missing value for " + feature)

case class FeatureFailedException(feature: Feature[\_], exception: Throwable) extends Exception

private[visibility] case class FeatureFailedPlaceholderObject(throwable: Throwable)

class FeatureMap(

val map: Map[Feature[\_], Stitch[\_]],

val constantMap: Map[Feature[\_], Any]) {

def contains[T](feature: Feature[T]): Boolean =

constantMap.contains(feature) || map.contains(feature)

def containsConstant[T](feature: Feature[T]): Boolean = constantMap.contains(feature)

lazy val size: Int = keys.size

lazy val keys: Set[Feature[\_]] = constantMap.keySet ++ map.keySet

def get[T](feature: Feature[T]): Stitch[T] = {

map.get(feature) match {

case \_ if constantMap.contains(feature) =>

Stitch.value(getConstant(feature))

case Some(x) =>

x.asInstanceOf[Stitch[T]]

case \_ =>

Stitch.exception(new MissingFeatureException(feature))

}

}

def getConstant[T](feature: Feature[T]): T = {

constantMap.get(feature) match {

case Some(x) =>

x.asInstanceOf[T]

case \_ =>

throw new MissingFeatureException(feature)

}

}

def -[T](key: Feature[T]): FeatureMap = new FeatureMap(map - key, constantMap - key)

override def toString: String = "FeatureMap(%s, %s)".format(map, constantMap)

}

object FeatureMap {

def empty: FeatureMap = new FeatureMap(Map.empty, Map.empty)

def resolve(

featureMap: FeatureMap,

statsReceiver: StatsReceiver = NullStatsReceiver

): Stitch[ResolvedFeatureMap] = {

val featureMapHydrationStatsReceiver = statsReceiver.scope("feature\_map\_hydration")

Stitch

.traverse(featureMap.map.toSeq) {

case (feature, value: Stitch[\_]) =>

val featureStatsReceiver = featureMapHydrationStatsReceiver.scope(feature.name)

lazy val featureFailureStat = featureStatsReceiver.scope("failures")

val featureStitch: Stitch[(Feature[\_], Any)] = value

.map { resolvedValue =>

featureStatsReceiver.counter("success").incr()

(feature, resolvedValue)

}

featureStitch

.handle {

case ffe: FeatureFailedException =>

featureFailureStat.counter().incr()

featureFailureStat.counter(ffe.exception.getClass.getName).incr()

(feature, FeatureFailedPlaceholderObject(ffe.exception))

}

.ensure {

featureStatsReceiver.counter("requests").incr()

}

}

.map { resolvedFeatures: Seq[(Feature[\_], Any)] =>

new ResolvedFeatureMap(resolvedFeatures.toMap ++ featureMap.constantMap)

}

}

def rescueFeatureTuple(kv: (Feature[\_], Stitch[\_])): (Feature[\_], Stitch[\_]) = {

val (k, v) = kv

val rescueValue = v.rescue {

case e =>

e match {

case cdre: ClientDiscardedRequestException => Stitch.exception(cdre)

case \_ => Stitch.exception(FeatureFailedException(k, e))

}

}

(k, rescueValue)

}

}

class ResolvedFeatureMap(private[visibility] val resolvedMap: Map[Feature[\_], Any])

extends FeatureMap(Map.empty, resolvedMap) {

override def equals(other: Any): Boolean = other match {

case otherResolvedFeatureMap: ResolvedFeatureMap =>

this.resolvedMap.equals(otherResolvedFeatureMap.resolvedMap)

case \_ => false

}

override def toString: String = "ResolvedFeatureMap(%s)".format(resolvedMap)

}

object ResolvedFeatureMap {

def apply(resolvedMap: Map[Feature[\_], Any]): ResolvedFeatureMap = {

new ResolvedFeatureMap(resolvedMap)

}

}