package com.twitter.product\_mixer.core.service.feature\_hydrator\_observer

import com.twitter.finagle.stats.BroadcastStatsReceiver

import com.twitter.finagle.stats.Counter

import com.twitter.finagle.stats.RollupStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.ml.featurestore.lib.data.HydrationError

import com.twitter.product\_mixer.core.feature.Feature

import com.twitter.product\_mixer.core.feature.featuremap.FeatureMap

import com.twitter.product\_mixer.core.feature.featurestorev1.featurevalue.FeatureStoreV1ResponseFeature

import com.twitter.product\_mixer.core.functional\_component.feature\_hydrator.FeatureHydrator

import com.twitter.product\_mixer.core.functional\_component.feature\_hydrator.featurestorev1.FeatureStoreV1CandidateFeatureHydrator

import com.twitter.product\_mixer.core.functional\_component.feature\_hydrator.featurestorev1.FeatureStoreV1QueryFeatureHydrator

import com.twitter.product\_mixer.core.model.common.identifier.ComponentIdentifier

import com.twitter.product\_mixer.core.service.Executor

import com.twitter.product\_mixer.shared\_library.observer.Observer

import com.twitter.servo.util.CancelledExceptionExtractor

import com.twitter.util.Throw

import com.twitter.util.Throwables

class FeatureHydratorObserver(

statsReceiver: StatsReceiver,

hydrators: Seq[FeatureHydrator[\_]],

context: Executor.Context) {

private val hydratorAndFeatureToStats: Map[

ComponentIdentifier,

Map[Feature[\_, \_], FeatureCounters]

] =

hydrators.map { hydrator =>

val hydratorScope = Executor.buildScopes(context, hydrator.identifier)

val featureToCounterMap: Map[Feature[\_, \_], FeatureCounters] = hydrator.features

.asInstanceOf[Set[Feature[\_, \_]]].map { feature =>

val scopedStats = scopedBroadcastStats(hydratorScope, feature)

// Initialize so we have them registered

val requestsCounter = scopedStats.counter(Observer.Requests)

val successCounter = scopedStats.counter(Observer.Success)

// These are dynamic so we can't really cache them

scopedStats.counter(Observer.Failures)

scopedStats.counter(Observer.Cancelled)

feature -> FeatureCounters(requestsCounter, successCounter, scopedStats)

}.toMap

hydrator.identifier -> featureToCounterMap

}.toMap

def observeFeatureSuccessAndFailures(

hydrator: FeatureHydrator[\_],

featureMaps: Seq[FeatureMap]

): Unit = {

val features = hydrator.features.asInstanceOf[Set[Feature[\_, \_]]]

val failedFeaturesWithErrorNames: Map[Feature[\_, \_], Seq[Seq[String]]] = hydrator match {

case \_: FeatureStoreV1QueryFeatureHydrator[\_] |

\_: FeatureStoreV1CandidateFeatureHydrator[\_, \_] =>

featureMaps.toIterator

.flatMap(\_.getTry(FeatureStoreV1ResponseFeature).toOption.map(\_.failedFeatures)).flatMap {

failureMap: Map[\_ <: Feature[\_, \_], Set[HydrationError]] =>

failureMap.flatMap {

case (feature, errors: Set[HydrationError]) =>

errors.headOption.map { error =>

feature -> Seq(Observer.Failures, error.errorType)

}

}.toIterator

}.toSeq.groupBy { case (feature, \_) => feature }.mapValues { seqOfTuples =>

seqOfTuples.map { case (\_, error) => error }

}

case \_: FeatureHydrator[\_] =>

features.toIterator

.flatMap { feature =>

featureMaps

.flatMap(\_.underlyingMap

.get(feature).collect {

case Throw(CancelledExceptionExtractor(throwable)) =>

(feature, Observer.Cancelled +: Throwables.mkString(throwable))

case Throw(throwable) =>

(feature, Observer.Failures +: Throwables.mkString(throwable))

})

}.toSeq.groupBy { case (feature, \_) => feature }.mapValues { seqOfTuples =>

seqOfTuples.map { case (\_, error) => error }

}

}

val failedFeaturesWithErrorCountsMap: Map[Feature[\_, \_], Map[Seq[String], Int]] =

failedFeaturesWithErrorNames.mapValues(\_.groupBy { statKey => statKey }.mapValues(\_.size))

val featuresToCounterMap = hydratorAndFeatureToStats.getOrElse(

hydrator.identifier,

throw new MissingHydratorException(hydrator.identifier))

features.foreach { feature =>

val hydratorFeatureCounters: FeatureCounters = featuresToCounterMap.getOrElse(

feature,

throw new MissingFeatureException(hydrator.identifier, feature))

val failedMapsCount = failedFeaturesWithErrorNames.getOrElse(feature, Seq.empty).size

val failedFeatureErrorCounts = failedFeaturesWithErrorCountsMap.getOrElse(feature, Map.empty)

hydratorFeatureCounters.requestsCounter.incr(featureMaps.size)

hydratorFeatureCounters.successCounter.incr(featureMaps.size - failedMapsCount)

failedFeatureErrorCounts.foreach {

case (failure, count) =>

hydratorFeatureCounters.scopedStats.counter(failure: \_\*).incr(count)

}

}

}

private def scopedBroadcastStats(

hydratorScope: Executor.Scopes,

feature: Feature[\_, \_],

): StatsReceiver = {

val suffix = Seq("Feature", feature.toString)

val localScope = hydratorScope.componentScopes ++ suffix

val relativeScope = hydratorScope.relativeScope ++ suffix

new RollupStatsReceiver(

BroadcastStatsReceiver(

Seq(

statsReceiver.scope(localScope: \_\*),

statsReceiver.scope(relativeScope: \_\*),

)

))

}

}

case class FeatureCounters(

requestsCounter: Counter,

successCounter: Counter,

scopedStats: StatsReceiver)

class MissingHydratorException(featureHydratorIdentifier: ComponentIdentifier)

extends Exception(s"Missing Feature Hydrator in Stats Map: ${featureHydratorIdentifier.name}")

class MissingFeatureException(

featureHydratorIdentifier: ComponentIdentifier,

feature: Feature[\_, \_])

extends Exception(

s"Missing Feature in Stats Map: ${feature.toString} for ${featureHydratorIdentifier.name}")