package com.twitter.product\_mixer.core.service.pipeline\_execution\_logger

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.inject.annotations.Flag

import com.twitter.product\_mixer.core.module.product\_mixer\_flags.ProductMixerFlagModule.PipelineExecutionLoggerAllowList

import com.twitter.product\_mixer.core.module.product\_mixer\_flags.ProductMixerFlagModule.ServiceLocal

import com.twitter.product\_mixer.core.pipeline.PipelineQuery

import com.twitter.product\_mixer.core.util.FuturePools

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

import com.twitter.util.Try

import com.twitter.util.logging.Logging

import pprint.PPrinter

import pprint.Tree

import pprint.Util

import pprint.tuplePrefix

import javax.inject.Inject

import javax.inject.Singleton

/\*\*

\* Initial implementation from:

\* https://stackoverflow.com/questions/15718506/scala-how-to-print-case-classes-like-pretty-printed-tree/57080463#57080463

\*/

object AllowListedPipelineExecutionLogger {

/\*\*

\* Given a case class who's arguments are all declared fields on the class,

\* returns an iterator of the field name and values

\*/

private[pipeline\_execution\_logger] def caseClassFields(

caseClass: Product

): Iterator[(String, Any)] = {

val fieldValues = caseClass.productIterator.toSet

val fields = caseClass.getClass.getDeclaredFields.toSeq

.filterNot(f => f.isSynthetic || java.lang.reflect.Modifier.isStatic(f.getModifiers))

fields.iterator

.map { f =>

f.setAccessible(true)

f.getName -> f.get(caseClass)

}.filter { case (\_, v) => fieldValues.contains(v) }

}

/\*\*

\* Returns whether a given [[Product]] is a case class which we can render nicely which:

\* - has a [[Product.productArity]] <= the number of declared fields

\* - isn't a built in binary operator

\* - isn't a tuple

\* - who's arguments are fields (not methods)

\* - every [[Product.productElement]] has a corresponding field

\*

\* This will return false for some case classes where we can not reliably determine which field names correspond to

\* each value in the case class (this can happen if a case class implements an abstract class resulting in val fields

\* becoming methods.

\*/

private[pipeline\_execution\_logger] def isRenderableCaseClass(caseClass: Product): Boolean = {

val possibleToBeRenderableCaseClass =

caseClass.getClass.getDeclaredFields.length >= caseClass.productArity

val isntBuiltInOperator =

!(caseClass.productArity == 2 && Util.isOperator(caseClass.productPrefix))

val isntTuple = !caseClass.getClass.getName.startsWith(tuplePrefix)

val declaredFieldsMatchesCaseClassFields = {

val caseClassFields = caseClass.productIterator.toSet

caseClass.getClass.getDeclaredFields.iterator

.filterNot(f => f.isSynthetic || java.lang.reflect.Modifier.isStatic(f.getModifiers))

.count { f =>

f.setAccessible(true)

caseClassFields.contains(f.get(caseClass))

} >= caseClass.productArity

}

possibleToBeRenderableCaseClass && isntBuiltInOperator && isntTuple && declaredFieldsMatchesCaseClassFields

}

/\*\* Makes a [[Tree]] which will render as `key = value` \*/

private def keyValuePair(key: String, value: Tree): Tree = {

Tree.Infix(Tree.Literal(key), "=", value)

}

/\*\*

\* Special handling for case classes who's field names can be easily paired with their values.

\* This will make the [[PPrinter]] render them as

\* {{{

\* CaseClassName(

\* field1 = value1,

\* field2 = value2

\* )

\* }}}

\* instead of

\* {{{

\* CaseClassName(

\* value1,

\* value2

\* )

\* }}}

\*

\* For case classes who's fields end up being compiled as methods, this will fall back

\* to the built in handling of case classes without their field names.

\*/

private[pipeline\_execution\_logger] def additionalHandlers: PartialFunction[Any, Tree] = {

case caseClass: Product if isRenderableCaseClass(caseClass) =>

Tree.Apply(

caseClass.productPrefix,

caseClassFields(caseClass).flatMap {

case (key, value) =>

val valueTree = printer.treeify(value, false, true)

Seq(keyValuePair(key, valueTree))

}

)

}

/\*\*

\* [[PPrinter]] instance to use when rendering scala objects

\* uses BlackAndWhite because colors mangle the output when looking at the logs in plain text

\*/

private val printer: PPrinter = PPrinter.BlackWhite.copy(

// arbitrary high value to turn off truncation

defaultHeight = Int.MaxValue,

// the relatively high width will cause some wrapping but many of the pretty printed objects

// will be sparse (e.g. None,\n None,\n, None,\n)

defaultWidth = 300,

// use reflection to print field names (can be deleted in Scala 2.13)

additionalHandlers = additionalHandlers

)

/\*\* Given any scala object, return a String representation of it \*/

private[pipeline\_execution\_logger] def objectAsString(o: Any): String =

printer.tokenize(o).mkString

}

@Singleton

class AllowListedPipelineExecutionLogger @Inject() (

@Flag(ServiceLocal) isServiceLocal: Boolean,

@Flag(PipelineExecutionLoggerAllowList) allowList: Seq[String],

statsReceiver: StatsReceiver)

extends PipelineExecutionLogger

with Logging {

private val scopedStats = statsReceiver.scope("AllowListedPipelineExecutionLogger")

val allowListRoles: Set[String] = allowList.toSet

private val futurePool =

FuturePools.boundedFixedThreadPool(

"AllowListedPipelineExecutionLogger",

// single thread, may need to be adjusted higher if it cant keep up with the work queue

fixedThreadCount = 1,

// arbitrarily large enough to handle spikes without causing large allocations or retaining past multiple GC cycles

workQueueSize = 100,

scopedStats

)

private val futureObserver = new FutureObserver[Unit](scopedStats, Seq.empty)

private val loggerOutputPath = Try(System.getProperty("log.allow\_listed\_execution\_logger.output"))

override def apply(pipelineQuery: PipelineQuery, message: Any): Unit = {

val userRoles: Set[String] = pipelineQuery.clientContext.userRoles.getOrElse(Set.empty)

// Check if this request is in the allowlist via a cleverly optimized set intersection

val allowListed =

if (allowListRoles.size > userRoles.size)

userRoles.exists(allowListRoles.contains)

else

allowListRoles.exists(userRoles.contains)

if (isServiceLocal || allowListed) {

futureObserver(

/\*\*

\* failure to enqueue the work will result with a failed [[com.twitter.util.Future]]

\* containing a [[java.util.concurrent.RejectedExecutionException]] which the wrapping [[futureObserver]]

\* will record metrics for.

\*/

futurePool {

logger.info(AllowListedPipelineExecutionLogger.objectAsString(message))

if (isServiceLocal && loggerOutputPath.isReturn) {

println(s"Logged request to: ${loggerOutputPath.get()}")

}

}

)

}

}

}