package com.twitter.product\_mixer.core.service

import com.twitter.finagle.stats.StatsReceiver

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

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

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

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

import com.twitter.product\_mixer.core.pipeline.pipeline\_failure.PipelineFailure

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

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

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

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

import com.twitter.util.Duration

import com.twitter.util.Return

import com.twitter.util.Throw

import com.twitter.util.Try

private[core] object ExecutorObserver {

/\*\* Make a [[ExecutorObserver]] with stats for the [[ComponentIdentifier]] and relative to the parent in the [[Context.componentStack]] \*/

def executorObserver[T](

context: Context,

currentComponentIdentifier: ComponentIdentifier,

statsReceiver: StatsReceiver

): ExecutorObserver[T] = new ExecutorObserver[T](

Executor.broadcastStatsReceiver(context, currentComponentIdentifier, statsReceiver))

/\*\* Make a [[ExecutorObserverWithSize]] with stats for the [[ComponentIdentifier]] and relative to the parent in the [[Context.componentStack]] \*/

def executorObserverWithSize(

context: Context,

currentComponentIdentifier: ComponentIdentifier,

statsReceiver: StatsReceiver

): ExecutorObserverWithSize = new ExecutorObserverWithSize(

Executor.broadcastStatsReceiver(context, currentComponentIdentifier, statsReceiver))

/\*\* Make a [[PipelineExecutorObserver]] with stats for the [[ComponentIdentifier]] and relative to the parent in the [[Context.componentStack]] \*/

def pipelineExecutorObserver[T <: PipelineResult[\_]](

context: Context,

currentComponentIdentifier: ComponentIdentifier,

statsReceiver: StatsReceiver

): PipelineExecutorObserver[T] = new PipelineExecutorObserver[T](

Executor.broadcastStatsReceiver(context, currentComponentIdentifier, statsReceiver))

/\*\*

\* Make a [[PipelineExecutorObserver]] specifically for a [[com.twitter.product\_mixer.core.pipeline.product.ProductPipeline]]

\* with no relative stats

\*/

def productPipelineExecutorObserver[T <: PipelineResult[\_]](

currentComponentIdentifier: ProductPipelineIdentifier,

statsReceiver: StatsReceiver

): PipelineExecutorObserver[T] =

new PipelineExecutorObserver[T](statsReceiver.scope(currentComponentIdentifier.toScopes: \_\*))

/\*\*

\* Make a [[PipelineExecutorObserver]] with only stats relative to the parent pipeline

\* for [[com.twitter.product\_mixer.core.pipeline.PipelineBuilder.Step]]s

\*/

def stepExecutorObserver(

context: Context,

currentComponentIdentifier: PipelineStepIdentifier,

statsReceiver: StatsReceiver

): ExecutorObserver[Unit] = {

new ExecutorObserver[Unit](

statsReceiver.scope(

Executor.buildScopes(context, currentComponentIdentifier).relativeScope: \_\*))

}

}

/\*\*

\* An [[Observer]] which is called as a side effect. Unlike the other observers which wrap a computation,

\* this [[Observer]] expects the caller to provide the latency value and wire it in

\*/

private[core] sealed class ExecutorObserver[T](

override val statsReceiver: StatsReceiver)

extends {

/\*\*

\* always empty because we expect an already scoped [[com.twitter.finagle.stats.BroadcastStatsReceiver]] to be passed in

\* @note uses early definitions [[https://docs.scala-lang.org/tutorials/FAQ/initialization-order.html]] to avoid null values for `scopes` in [[Observer]]

\*/

override val scopes: Seq[String] = Seq.empty

} with Observer[T] {

/\*\*

\* Serialize the provided [[Throwable]], prefixing [[PipelineFailure]]s with their

\* [[com.twitter.product\_mixer.core.pipeline.pipeline\_failure.PipelineFailureCategory.categoryName]] and

\* [[com.twitter.product\_mixer.core.pipeline.pipeline\_failure.PipelineFailureCategory.failureName]]

\*/

override def serializeThrowable(throwable: Throwable): Seq[String] = {

throwable match {

case PipelineFailure(category, \_, None, \_) =>

Seq(category.categoryName, category.failureName)

case PipelineFailure(category, \_, Some(underlying), \_) =>

Seq(category.categoryName, category.failureName) ++ serializeThrowable(underlying)

case throwable: Throwable => super.serializeThrowable(throwable)

}

}

/\*\* record success, failure, and latency stats based on `t` and `latency` \*/

def apply(t: Try[T], latency: Duration): Unit = observe(t, latency)

}

/\*\*

\* Same as [[ExecutorObserver]] but records a size stat for [[PipelineResult]]s and

\* records a failure counter for the cause of the failure under `failures/$pipelineFailureCategory/$componentType/$componentName`.

\*

\* @example if `GateIdentifier("MyGate")` is at the top of the [[PipelineFailure.componentStack]] then

\* the resulting metric `failures/ClientFailure/Gate/MyGate` will be incremented.

\*/

private[core] final class PipelineExecutorObserver[T <: PipelineResult[\_]](

override val statsReceiver: StatsReceiver)

extends ExecutorObserver[T](statsReceiver)

with ResultsStatsObserver[T] {

override val size: T => Int = \_.resultSize()

override def apply(t: Try[T], latency: Duration): Unit = {

super.apply(t, latency)

t match {

case Return(result) => observeResults(result)

case Throw(PipelineFailure(category, \_, \_, Some(componentIdentifierStack))) =>

statsReceiver

.counter(

Seq(

Observer.Failures,

category.categoryName,

category.failureName) ++ componentIdentifierStack.peek.toScopes: \_\*).incr()

case \_ =>

}

}

}

/\*\* Same as [[ExecutorObserver]] but records a size stat \*/

private[core] final class ExecutorObserverWithSize(

override val statsReceiver: StatsReceiver)

extends ExecutorObserver[Int](statsReceiver)

with ResultsStatsObserver[Int] {

override val size: Int => Int = identity

override def apply(t: Try[Int], latency: Duration): Unit = {

super.apply(t, latency)

t match {

case Return(result) => observeResults(result)

case \_ =>

}

}

}