package com.twitter.product\_mixer.core.service.pipeline\_result\_side\_effect\_executor

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.product\_mixer.core.functional\_component.side\_effect.ExecuteSynchronously

import com.twitter.product\_mixer.core.functional\_component.side\_effect.FailOpen

import com.twitter.product\_mixer.core.functional\_component.side\_effect.PipelineResultSideEffect

import com.twitter.product\_mixer.core.functional\_component.side\_effect.PipelineResultSideEffect.Inputs

import com.twitter.product\_mixer.core.model.common.Conditionally

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

import com.twitter.product\_mixer.core.model.marshalling.HasMarshalling

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

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

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

import com.twitter.product\_mixer.core.service.pipeline\_result\_side\_effect\_executor.PipelineResultSideEffectExecutor.\_

import com.twitter.stitch.Arrow

import com.twitter.util.Return

import com.twitter.util.Try

import javax.inject.Inject

import javax.inject.Singleton

@Singleton

class PipelineResultSideEffectExecutor @Inject() (override val statsReceiver: StatsReceiver)

extends Executor {

def arrow[Query <: PipelineQuery, MixerDomainResultType <: HasMarshalling](

sideEffects: Seq[PipelineResultSideEffect[Query, MixerDomainResultType]],

context: Executor.Context

): Arrow[Inputs[Query, MixerDomainResultType], PipelineResultSideEffectExecutor.Result] = {

val individualArrows: Seq[

Arrow[Inputs[Query, MixerDomainResultType], (SideEffectIdentifier, SideEffectResultType)]

] = sideEffects.map {

case synchronousSideEffect: ExecuteSynchronously =>

val failsRequestIfThrows = {

wrapComponentWithExecutorBookkeeping(context, synchronousSideEffect.identifier)(

Arrow.flatMap(synchronousSideEffect.apply))

}

synchronousSideEffect match {

case failOpen: FailOpen =>

// lift the failure

failsRequestIfThrows.liftToTry.map(t =>

(failOpen.identifier, SynchronousSideEffectResult(t)))

case \_ =>

// don't encapsulate the failure

failsRequestIfThrows.map(\_ =>

(synchronousSideEffect.identifier, SynchronousSideEffectResult(Return.Unit)))

}

case sideEffect =>

Arrow

.async(

wrapComponentWithExecutorBookkeeping(context, sideEffect.identifier)(

Arrow.flatMap(sideEffect.apply)))

.andThen(Arrow.value((sideEffect.identifier, SideEffectResult)))

}

val conditionallyRunArrows = sideEffects.zip(individualArrows).map {

case (

sideEffect: Conditionally[

PipelineResultSideEffect.Inputs[Query, MixerDomainResultType] @unchecked

],

arrow) =>

Arrow.ifelse[

Inputs[Query, MixerDomainResultType],

(SideEffectIdentifier, SideEffectResultType)](

input => sideEffect.onlyIf(input),

arrow,

Arrow.value((sideEffect.identifier, TurnedOffByConditionally)))

case (\_, arrow) => arrow

}

Arrow

.collect(conditionallyRunArrows)

.map(results => Result(results))

}

}

object PipelineResultSideEffectExecutor {

case class Result(sideEffects: Seq[(SideEffectIdentifier, SideEffectResultType)])

extends ExecutorResult

sealed trait SideEffectResultType

/\*\* The [[PipelineResultSideEffect]] was executed asynchronously in a fire-and-forget way so no result will be available \*/

case object SideEffectResult extends SideEffectResultType

/\*\* The result of the [[PipelineResultSideEffect]] that was executed with [[ExecuteSynchronously]] \*/

case class SynchronousSideEffectResult(result: Try[Unit]) extends SideEffectResultType

/\*\* The result for when a [[PipelineResultSideEffect]] is turned off by [[Conditionally]]'s [[Conditionally.onlyIf]] \*/

case object TurnedOffByConditionally extends SideEffectResultType

}