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

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.PipelineStepIdentifier

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

import com.twitter.product\_mixer.core.pipeline.state.HasExecutorResults

import com.twitter.product\_mixer.core.pipeline.state.HasResult

import com.twitter.product\_mixer.core.pipeline.step.Step

import com.twitter.product\_mixer.core.quality\_factor.QualityFactorStatus

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

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

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

import com.twitter.stitch.Arrow

import com.twitter.stitch.Arrow.Iso

import com.twitter.util.Return

import com.twitter.util.Throw

/\*\*

\* Pipeline Arrow Builder used for constructing a final arrow for a pipeline after adding necessary

\* steps.

\*

\* @param steps The kept non-empty Pipeline Steps

\* @param addedSteps Steps that have been added, but not necessarily kept.

\* @param statsReceiver Stats Receiver for metric book keeping

\* @tparam Result sThe expected final result type of the pipeline.

\* @tparam State The input state type, which should implement [[HasResult]].

\*/

case class NewPipelineArrowBuilder[

Result,

State <: HasExecutorResults[State] with HasResult[Result]

] private (

private val steps: Seq[PipelineStep[State, \_, \_, \_]],

override val statsReceiver: StatsReceiver)

extends Executor {

def add[Config, ExecutorInput, ExResult <: ExecutorResult](

pipelineStepIdentifier: PipelineStepIdentifier,

step: Step[State, Config, ExecutorInput, ExResult],

executorConfig: Config

): NewPipelineArrowBuilder[Result, State] = {

require(

!steps.contains(pipelineStepIdentifier),

s"Found duplicate step $pipelineStepIdentifier when building pipeline arrow")

// If the step has nothing to execute, drop it for simplification but still added it to the

// "addedSteps" field for build time validation

if (step.isEmpty(executorConfig)) {

this

} else {

val newPipelineStep =

PipelineStep(pipelineStepIdentifier, executorConfig, step)

val newSteps = steps :+ newPipelineStep

this.copy(steps = newSteps)

}

}

def buildArrow(

context: Executor.Context

): Arrow[State, NewPipelineResult[Result]] = {

val initialArrow = Arrow

.map { input: State => NewStepData[State](input) }

val allStepArrows = steps.map { step =>

Iso.onlyIf[NewStepData[State]] { stepData => !stepData.stopExecuting } {

wrapStepWithExecutorBookkeeping(step, context)

}

}

val combinedArrow = isoArrowsSequentially(allStepArrows)

val resultArrow = Arrow.map { stepData: NewStepData[State] =>

stepData.pipelineFailure match {

case Some(failure) =>

NewPipelineResult.Failure(failure, stepData.pipelineState.executorResultsByPipelineStep)

case None =>

NewPipelineResult.Success(

stepData.pipelineState.buildResult,

stepData.pipelineState.executorResultsByPipelineStep)

}

}

initialArrow.andThen(combinedArrow).andThen(resultArrow)

}

private[this] def wrapStepWithExecutorBookkeeping(

step: PipelineStep[State, \_, \_, \_],

context: Context

): Arrow.Iso[NewStepData[State]] = {

val wrapped = wrapStepWithExecutorBookkeeping[NewStepData[State], NewStepData[State]](

context = context,

identifier = step.stepIdentifier,

arrow = step.arrow(context),

// extract the failure only if it's present. Not sure if this is needed???

transformer = \_.pipelineFailure.map(Throw(\_)).getOrElse(Return.Unit)

)

Arrow

.zipWithArg(wrapped.liftToTry)

.map {

case (\_: NewStepData[State], Return(result)) =>

// if Step was successful, return the result

result

case (previous: NewStepData[State], Throw(pipelineFailure: PipelineFailure)) =>

// if the Step failed in such a way that the failure was NOT captured

// in the result object, then update the State with the failure

previous.withFailure(pipelineFailure)

case (\_, Throw(ex)) =>

// an exception was thrown which was not handled by the failure classifier

// this only happens with cancellation exceptions which are re-thrown

throw ex

}

}

/\*\*

\* Sets up stats [[com.twitter.finagle.stats.Gauge]]s for any [[QualityFactorStatus]]

\*

\* @note We use provideGauge so these gauges live forever even without a reference.

\*/

private[pipeline] def buildGaugesForQualityFactor(

pipelineIdentifier: ComponentIdentifier,

qualityFactorStatus: QualityFactorStatus,

statsReceiver: StatsReceiver

): Unit = {

qualityFactorStatus.qualityFactorByPipeline.foreach {

case (identifier, qualityFactor) =>

// QF is a relative stat (since the parent pipeline is monitoring a child pipeline)

val scopes = pipelineIdentifier.toScopes ++ identifier.toScopes :+ "QualityFactor"

statsReceiver.provideGauge(scopes: \_\*) { qualityFactor.currentValue.toFloat }

}

}

}

object NewPipelineArrowBuilder {

def apply[Result, InputState <: HasExecutorResults[InputState] with HasResult[Result]](

statsReceiver: StatsReceiver

): NewPipelineArrowBuilder[Result, InputState] = {

NewPipelineArrowBuilder(

Seq.empty,

statsReceiver

)

}

}

/\*\*

\* This is a pipeline specific instance of a step, i.e, a generic step with the step identifier

\* within the pipeline and its executor configs.

\* @param stepIdentifier Step identifier of the step within a pipeline

\* @param executorConfig Config to execute the step with

\* @param step The underlying step to be used

\* @tparam InputState The input state object

\* @tparam ExecutorConfig The config expected for the given step

\* @tparam ExecutorInput Input for the underlying executor

\* @tparam ExecResult The result type

\*/

case class PipelineStep[

State <: HasExecutorResults[State],

PipelineStepConfig,

ExecutorInput,

ExecResult <: ExecutorResult

](

stepIdentifier: PipelineStepIdentifier,

executorConfig: PipelineStepConfig,

step: Step[State, PipelineStepConfig, ExecutorInput, ExecResult]) {

def arrow(

context: Executor.Context

): Arrow.Iso[NewStepData[State]] = {

val inputArrow = Arrow.map { stepData: NewStepData[State] =>

step.adaptInput(stepData.pipelineState, executorConfig)

}

Arrow

.zipWithArg(inputArrow.andThen(step.arrow(executorConfig, context))).map {

case (stepData: NewStepData[State], executorResult: ExecResult @unchecked) =>

val updatedResultsByPipelineStep =

stepData.pipelineState.executorResultsByPipelineStep + (stepIdentifier -> executorResult)

val updatedPipelineState = step

.updateState(stepData.pipelineState, executorResult, executorConfig).setExecutorResults(

updatedResultsByPipelineStep)

NewStepData(updatedPipelineState)

}

}

}