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.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

import com.twitter.stitch.Arrow

import com.twitter.util.Return

import com.twitter.util.Throw

trait PipelineBuilder[Query] extends Executor {

/\*\*

\* When a step is mostly the same, but only the result update changes,

\* you can pass in a [[ResultUpdater]] to the [[Step]] to perform the update

\* such as with multi-step hydration.

\*/

trait ResultUpdater[R <: PipelineResult[\_], ER <: service.ExecutorResult] {

def apply(existingResult: R, executorResult: ER): R

}

type UnderlyingResultType

type PipelineResultType <: PipelineResult[UnderlyingResultType]

/\*\* the data that every step has as input and output - the query, and the in-progress result \*/

case class StepData(query: Query, result: PipelineResultType)

/\*\* An [[Arrow.Iso]] [[Arrow]] is an arrow with the same input and output types. \*/

type StepArrow = Arrow.Iso[StepData]

/\*\*

\* We break pipeline execution into a linear sequence of [[Step]]s. The execution logic of each

\* step is represented as an [[Executor]] (which is reusable between pipelines).

\*

\* Each step has access to the [[PipelineResult]] generated by previous steps, and can update it

\* with some new data.

\*

\* We define a pipeline Step as having three parts:

\*

\* - An underlying [[Executor]] [[Arrow]], from the underlying executor

\* - An input adaptor to extract the right data from the previous [[PipelineResult]]

\* - A result updater to update the [[PipelineResult]]

\*

\* This keeps knowledge of [[PipelineResult]] out of the executors, so they're reusable.

\*

\* @tparam ExecutorInput The input type used by the executor

\* @tparam ExecutorResult The output/result type used by the executor

\*/

trait Step[ExecutorInput, ExecutorResult] {

def identifier: PipelineStepIdentifier

def executorArrow: Arrow[ExecutorInput, ExecutorResult]

def inputAdaptor(query: Query, previousResult: PipelineResultType): ExecutorInput

def resultUpdater(

previousPipelineResult: PipelineResultType,

executorResult: ExecutorResult

): PipelineResultType

/\*\*

\* Optionally, steps can define a function to update the Query

\*/

def queryUpdater(query: Query, executorResult: ExecutorResult): Query = query

/\*\*

\* Arrow that adapts the input, runs the underlying Executor, adapts the output, and updates the state

\*/

val stepArrow: StepArrow = {

val inputAdaptorArrow: Arrow[StepData, ExecutorInput] = Arrow.map { stepData: StepData =>

inputAdaptor(stepData.query, stepData.result)

}

val outputAdaptorArrow: Arrow[(StepData, ExecutorResult), StepData] = Arrow.map {

// abstract type pattern ExecutorResult is unchecked since it is eliminated by erasure

case (previousStepData: StepData, executorResult: ExecutorResult @unchecked) =>

StepData(

query = queryUpdater(previousStepData.query, executorResult),

result = resultUpdater(previousStepData.result, executorResult)

)

}

Arrow

.zipWithArg(inputAdaptorArrow.andThen(executorArrow))

.andThen(outputAdaptorArrow)

}

}

/\*\*

\* Wraps a step with [[wrapStepWithExecutorBookkeeping]]

\*

\* When an error is encountered in execution, we update the [[PipelineResult.failure]] field,

\* and we return the partial results from all previously executed steps.

\*/

def wrapStepWithExecutorBookkeeping(

context: Context,

step: Step[\_, \_]

): Arrow.Iso[StepData] = {

val wrapped = wrapStepWithExecutorBookkeeping[StepData, StepData](

context = context,

identifier = step.identifier,

arrow = step.stepArrow,

// extract the failure only if it's present

transformer = \_.result.failure match {

case Some(pipelineFailure) => Throw(pipelineFailure)

case None => Return.Unit

}

)

Arrow

.zipWithArg(wrapped.liftToTry)

.map {

case (\_: StepData, Return(result)) =>

// if Step was successful, return the result

result

case (StepData(query, previousResult), 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

StepData(

query,

previousResult.withFailure(pipelineFailure).asInstanceOf[PipelineResultType])

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

}

}

/\*\*

\* Builds a combined arrow out of steps.

\*

\* Wraps them in error handling, and only executes each step if the previous step is successful.

\*/

def buildCombinedArrowFromSteps(

steps: Seq[Step[\_, \_]],

context: Executor.Context,

initialEmptyResult: PipelineResultType,

stepsInOrderFromConfig: Seq[PipelineStepIdentifier]

): Arrow[Query, PipelineResultType] = {

validateConfigAndBuilderAreInSync(steps, stepsInOrderFromConfig)

/\*\*

\* Prepare the step arrows.

\* 1. Wrap them in executor bookkeeping

\* 2. Wrap them in Iso.onlyIf - so we only execute them if we don't have a result or failure yet

\* 3. Combine them using [[isoArrowsSequentially]]

\*

\* @note this results in no Executor bookkeeping actions for [[Step]]s after

\* we reach a [[PipelineResult.stopExecuting]].

\*/

val stepArrows = isoArrowsSequentially(steps.map { step =>

Arrow.Iso.onlyIf[StepData](stepData => !stepData.result.stopExecuting)(

wrapStepWithExecutorBookkeeping(context, step))

})

Arrow

.identity[Query]

.map { query => StepData(query, initialEmptyResult) }

.andThen(stepArrows)

.map { case StepData(\_, result) => result }

}

/\*\*

\* 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 }

}

}

/\*\* Validates that the [[PipelineConfigCompanion]] is in sync with the [[Step]]s a [[PipelineBuilder]] produces \*/

private[this] def validateConfigAndBuilderAreInSync(

builtSteps: Seq[Step[\_, \_]],

stepsInOrder: Seq[PipelineStepIdentifier]

): Unit = {

require(

builtSteps.map(\_.identifier) == stepsInOrder,

s"Builder and Config are out of sync, bug in Product Mixer Core, `PipelineCompanion` and `PipelineBuilder` " +

s"have different definitions of what Steps are run in this Pipeline \n" +

s"${builtSteps.map(\_.identifier).zip(stepsInOrder).mkString("\n")}"

)

}

}