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

import com.twitter.finagle.stats.BroadcastStatsReceiver

import com.twitter.finagle.stats.Counter

import com.twitter.finagle.stats.DefaultStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.tracing.Annotation

import com.twitter.finagle.tracing.Record

import com.twitter.finagle.tracing.Trace

import com.twitter.finagle.tracing.TraceId

import com.twitter.finagle.tracing.TraceServiceName

import com.twitter.finagle.tracing.Tracing.LocalBeginAnnotation

import com.twitter.finagle.tracing.Tracing.LocalEndAnnotation

import com.twitter.product\_mixer.core.feature.Feature

import com.twitter.product\_mixer.core.feature.featuremap.FeatureMap

import com.twitter.product\_mixer.core.feature.featuremap.FeatureMapBuilder

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

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

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

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.FailOpenPolicy

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

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

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

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

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

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

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

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

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

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

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

import com.twitter.servo.util.CancelledExceptionExtractor

import com.twitter.stitch.Arrow

import com.twitter.stitch.Stitch

import com.twitter.stitch.Stitch.Letter

import com.twitter.util.Duration

import com.twitter.util.Return

import com.twitter.util.Throw

import com.twitter.util.Time

import com.twitter.util.Try

/\*\*

\* Base trait that all executors implement

\*

\* All executors should:

\* - implement a `def arrow` or `def apply` with the relevant types for their use case

\* and take in an implicit [[PipelineFailureClassifier]] and [[ComponentIdentifierStack]].

\* - add a `@singleton` annotation to the class and `@inject` annotation to the argument list

\* - take in a [[StatsReceiver]]

\*

\* @example {{{

\* @Singleton class MyExecutor @Inject() (

\* override val statsReceiver: StatsReceiver

\* ) extends Executor {

\* def arrow(

\* arg: MyArg,

\* ...,

\* context: Context

\* ): Arrow[In,Out] = ???

\* }

\* }}}

\*/

private[core] trait Executor {

val statsReceiver: StatsReceiver

/\*\*

\* Applies the `pipelineFailureClassifier` to the output of the `arrow`

\* and adds the `componentStack` to the [[PipelineFailure]]

\*/

def wrapWithErrorHandling[In, Out](

context: Context,

currentComponentIdentifier: ComponentIdentifier

)(

arrow: Arrow[In, Out]

): Arrow[In, Out] = {

arrow.mapFailure(

toPipelineFailureWithComponentIdentifierStack(context, currentComponentIdentifier))

}

/\*\*

\* Chain a `Seq` of [[Arrow.Iso]], only passing successful results to the next [[Arrow.Iso]]

\*

\* @note the resulting [[Arrow]] runs the passed in [[Arrow]]s one after the other,

\* as an ordered execution, this means that each [[Arrow]] is dependent

\* on all previous [[Arrow]]s in the `Seq` so no `Stitch` batching can occur

\* between them.

\*/

def isoArrowsSequentially[T](arrows: Seq[Arrow.Iso[T]]): Arrow.Iso[T] = {

// avoid excess Arrow complexity when there is only a single Arrow

arrows match {

case Seq() => Arrow.identity

case Seq(onlyOneArrow) => onlyOneArrow

case Seq(head, tail @ \_\*) =>

tail.foldLeft(head) {

case (combinedArrow, nextArrow) => combinedArrow.flatMapArrow(nextArrow)

}

}

}

/\*\*

\* Start running the [[Arrow]] in the background returning a [[Stitch.Ref]] which will complete

\* when the background task is finished

\*/

def startArrowAsync[In, Out](arrow: Arrow[In, Out]): Arrow[In, Stitch[Out]] = {

Arrow

.map { arg: In =>

// wrap in a `ref` so we only compute it's value once

Stitch.ref(arrow(arg))

}

.andThen(

Arrow.zipWithArg(

// satisfy the `ref` async

Arrow.async(Arrow.flatMap[Stitch[Out], Out](identity))))

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

}

/\*\*

\* for [[com.twitter.product\_mixer.core.model.common.Component]]s which

\* are executed per-candidate or which we don't want to record stats for.

\* This performs Tracing but does not record Stats

\*

\* @note This should be used around the computation that includes the execution of the

\* underlying Component over all the Candidates, not around each execution

\* of the component around each candidate for per-candidate Components.

\*

\* @note when using this you should only use [[wrapPerCandidateComponentWithExecutorBookkeepingWithoutTracing]]

\* for handling Stats.

\*/

def wrapComponentsWithTracingOnly[In, Out](

context: Context,

currentComponentIdentifier: ComponentIdentifier

)(

arrow: Arrow[In, Out]

): Arrow[In, Out] = {

Executor.wrapArrowWithLocalTracingSpan(

Arrow

.time(arrow)

.map {

case (result, latency) =>

Executor.recordTraceData(

componentStack = context.componentStack,

componentIdentifier = currentComponentIdentifier,

result = result,

latency = latency,

size = None)

result

}.lowerFromTry)

}

/\*\*

\* for [[com.twitter.product\_mixer.core.model.common.Component]]s which

\* are executed per-candidate. Records Stats but does not do Tracing.

\*

\* @note when using this you should only use [[wrapPerCandidateComponentsWithTracingOnly]]

\* for handling Tracing

\*/

def wrapPerCandidateComponentWithExecutorBookkeepingWithoutTracing[In, Out](

context: Context,

currentComponentIdentifier: ComponentIdentifier

)(

arrow: Arrow[In, Out]

): Arrow[In, Out] = {

val observerSideEffect =

ExecutorObserver.executorObserver[Out](context, currentComponentIdentifier, statsReceiver)

Executor.wrapWithExecutorBookkeeping[In, Out, Out](

context = context,

currentComponentIdentifier = currentComponentIdentifier,

executorResultSideEffect = observerSideEffect,

transformer = Return(\_),

tracingConfig = TracingConfig.NoTracing

)(arrow)

}

/\*\* for [[com.twitter.product\_mixer.core.model.common.Component]]s \*/

def wrapComponentWithExecutorBookkeeping[In, Out](

context: Context,

currentComponentIdentifier: ComponentIdentifier

)(

arrow: Arrow[In, Out]

): Arrow[In, Out] = {

val observerSideEffect =

ExecutorObserver.executorObserver[Out](context, currentComponentIdentifier, statsReceiver)

Executor.wrapWithExecutorBookkeeping[In, Out, Out](

context = context,

currentComponentIdentifier = currentComponentIdentifier,

executorResultSideEffect = observerSideEffect,

transformer = Return(\_)

)(arrow)

}

/\*\*

\* for [[com.twitter.product\_mixer.core.model.common.Component]]s which an `onSuccess`

\* to add custom stats or logging of results

\*/

def wrapComponentWithExecutorBookkeeping[In, Out](

context: Context,

currentComponentIdentifier: ComponentIdentifier,

onSuccess: Out => Unit

)(

arrow: Arrow[In, Out]

): Arrow[In, Out] = {

val observerSideEffect =

ExecutorObserver.executorObserver[Out](context, currentComponentIdentifier, statsReceiver)

Executor.wrapWithExecutorBookkeeping[In, Out, Out](

context = context,

currentComponentIdentifier = currentComponentIdentifier,

executorResultSideEffect = observerSideEffect,

transformer = Return(\_),

onComplete = (transformed: Try[Out]) => transformed.onSuccess(onSuccess)

)(arrow)

}

/\*\* for [[com.twitter.product\_mixer.core.pipeline.Pipeline]]s \*/

def wrapPipelineWithExecutorBookkeeping[In, Out <: PipelineResult[\_]](

context: Context,

currentComponentIdentifier: ComponentIdentifier,

qualityFactorObserver: Option[QualityFactorObserver],

failOpenPolicy: FailOpenPolicy = FailOpenPolicy.Never

)(

arrow: Arrow[In, Out]

): Arrow[In, Out] = {

val observerSideEffect =

ExecutorObserver

.pipelineExecutorObserver[Out](context, currentComponentIdentifier, statsReceiver)

Executor.wrapWithExecutorBookkeeping[In, Out, Out](

context = context,

currentComponentIdentifier = currentComponentIdentifier,

executorResultSideEffect = observerSideEffect,

transformer = (result: Out) => result.toTry,

size = Some(\_.resultSize()),

failOpenPolicy = failOpenPolicy,

qualityFactorObserver = qualityFactorObserver

)(arrow)

}

/\*\* for [[com.twitter.product\_mixer.core.pipeline.product.ProductPipeline]]s \*/

def wrapProductPipelineWithExecutorBookkeeping[In, Out <: PipelineResult[\_]](

context: Context,

currentComponentIdentifier: ProductPipelineIdentifier

)(

arrow: Arrow[In, Out]

): Arrow[In, Out] = {

val observerSideEffect =

ExecutorObserver

.productPipelineExecutorObserver[Out](currentComponentIdentifier, statsReceiver)

Executor.wrapWithExecutorBookkeeping[In, Out, Out](

context = context,

currentComponentIdentifier = currentComponentIdentifier,

executorResultSideEffect = observerSideEffect,

transformer = \_.toTry,

size = Some(\_.resultSize()),

failOpenPolicy =

// always save Failures in the Result object instead of failing the request

AlwaysFailOpenIncludingProgrammerErrors

)(arrow)

}

/\*\* for [[com.twitter.product\_mixer.core.model.common.Component]]s which need a result size stat \*/

def wrapComponentWithExecutorBookkeepingWithSize[In, Out](

context: Context,

currentComponentIdentifier: CandidateSourceIdentifier,

size: Out => Int

)(

arrow: Arrow[In, Out]

): Arrow[In, Out] = {

val observerSideEffect =

ExecutorObserver.executorObserverWithSize(context, currentComponentIdentifier, statsReceiver)

Executor.wrapWithExecutorBookkeeping[In, Out, Int](

context = context,

currentComponentIdentifier = currentComponentIdentifier,

executorResultSideEffect = observerSideEffect,

transformer = (out: Out) => Try(size(out)),

size = Some(identity)

)(arrow)

}

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

def wrapStepWithExecutorBookkeeping[In, Out](

context: Context,

identifier: PipelineStepIdentifier,

arrow: Arrow[In, Out],

transformer: Out => Try[Unit]

): Arrow[In, Out] = {

val observerSideEffect =

ExecutorObserver.stepExecutorObserver(context, identifier, statsReceiver)

Executor.wrapWithExecutorBookkeeping[In, Out, Unit](

context = context,

currentComponentIdentifier = identifier,

executorResultSideEffect = observerSideEffect,

transformer = transformer,

failOpenPolicy = AlwaysFailOpenIncludingProgrammerErrors

)(arrow)

}

}

private[core] object Executor {

private[service] object TracingConfig {

/\*\* Used to specify whether a wrapped Arrow should be Traced in [[wrapWithExecutorBookkeeping]] \*/

sealed trait TracingConfig

case object NoTracing extends TracingConfig

case object WrapWithSpanAndTracingData extends TracingConfig

}

/\*\*

\* Always fail-open and return the [[com.twitter.product\_mixer.core.pipeline.product.ProductPipelineResult]]

\* containing the exception, this differs from [[FailOpenPolicy.Always]] in that this will still

\* fail-open and return the overall result object even if the underlying failure is the result

\* of programmer error.

\*/

private val AlwaysFailOpenIncludingProgrammerErrors: FailOpenPolicy = \_ => true

/\*\*

\* Wraps an [[Arrow]] so that bookkeeping around the execution occurs uniformly.

\*

\* @note should \_\_never\_\_ be called directly!

\*

\* - For successful results, apply the `transformer`

\* - convert any exceptions to PipelineFailures

\* - record stats and update [[QualityFactorObserver]]

\* - wraps the execution in a Trace span and record Trace data (can be turned off by [[TracingConfig]])

\* - applies a trace span and records metadata to the provided `arrow`

\* - determine whether to fail-open based on `result.flatMap(transformer)`

\* - if failing-open, always return the original result

\* - if failing-closed and successful, return the original result

\* - otherwise, return the failure (from `result.flatMap(transformer)`)

\*

\* @param context the [[Executor.Context]]

\* @param currentComponentIdentifier the current component's [[ComponentIdentifier]]

\* @param executorResultSideEffect the [[ExecutorObserver]] used to record stats

\* @param transformer function to convert a successful result into possibly a failed result

\* @param failOpenPolicy [[FailOpenPolicy]] to apply to the results of `result.flatMap(transformer)`

\* @param qualityFactorObserver [[QualityFactorObserver]] to update based on the results of `result.flatMap(transformer)`

\* @param tracingConfig indicates whether the [[Arrow]] should be wrapped with Tracing

\* @param onComplete runs the function for its side effects with the result of `result.flatMap(transformer)`

\* @param arrow an input [[Arrow]] to wrap so that after it's execution, we perform all these operations

\*

\* @return the wrapped [[Arrow]]

\*/

private[service] def wrapWithExecutorBookkeeping[In, Out, Transformed](

context: Context,

currentComponentIdentifier: ComponentIdentifier,

executorResultSideEffect: ExecutorObserver[Transformed],

transformer: Out => Try[Transformed],

size: Option[Transformed => Int] = None,

failOpenPolicy: FailOpenPolicy = FailOpenPolicy.Never,

qualityFactorObserver: Option[QualityFactorObserver] = None,

tracingConfig: TracingConfig.TracingConfig = TracingConfig.WrapWithSpanAndTracingData,

onComplete: Try[Transformed] => Unit = { \_: Try[Transformed] => () }

)(

arrow: Arrow[In, Out]

): Arrow[In, Out] = {

val failureClassifier =

toPipelineFailureWithComponentIdentifierStack(context, currentComponentIdentifier)

/\*\* transform the results, mapping all exceptions to [[PipelineFailure]]s, and tuple with original result \*/

val transformResultAndClassifyFailures: Arrow[Out, (Out, Try[Transformed])] =

Arrow.join(

Arrow.mapFailure(failureClassifier),

Arrow

.transformTry[Out, Transformed](result =>

result

.flatMap(transformer)

.rescue { case t => Throw(failureClassifier(t)) })

.liftToTry

)

/\*\* Only record tracing data if [[TracingConfig.WrapWithSpanAndTracingData]] \*/

val maybeRecordTracingData: (Try[Transformed], Duration) => Unit = tracingConfig match {

case TracingConfig.NoTracing => (\_, \_) => ()

case TracingConfig.WrapWithSpanAndTracingData =>

(transformedAndClassifiedResult, latency) =>

recordTraceData(

context.componentStack,

currentComponentIdentifier,

transformedAndClassifiedResult,

latency,

transformedAndClassifiedResult.toOption.flatMap(result => size.map(\_.apply(result)))

)

}

/\*\* Will never be in a failed state so we can do a simple [[Arrow.map]] \*/

val recordStatsAndUpdateQualityFactor =

Arrow

.map[(Try[(Out, Try[Transformed])], Duration), Unit] {

case (tryResultAndTryTransformed, latency) =>

val transformedAndClassifiedResult = tryResultAndTryTransformed.flatMap {

case (\_, transformed) => transformed

}

executorResultSideEffect(transformedAndClassifiedResult, latency)

qualityFactorObserver.foreach(\_.apply(transformedAndClassifiedResult, latency))

onComplete(transformedAndClassifiedResult)

maybeRecordTracingData(transformedAndClassifiedResult, latency)

}.unit

/\*\*

\* Applies the provided [[FailOpenPolicy]] based on the [[transformer]]'s results,

\* returning the original result or an exception

\*/

val applyFailOpenPolicyBasedOnTransformedResult: Arrow[

(Try[(Out, Try[Transformed])], Duration),

Out

] =

Arrow

.map[(Try[(Out, Try[Transformed])], Duration), Try[(Out, Try[Transformed])]] {

case (tryResultAndTryTransformed, \_) => tryResultAndTryTransformed

}

.lowerFromTry

.map {

case (result, Throw(pipelineFailure: PipelineFailure))

if failOpenPolicy(pipelineFailure.category) =>

Return(result)

case (\_, t: Throw[\_]) => t.asInstanceOf[Throw[Out]]

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

}.lowerFromTry

/\*\* The complete Arrow minus a Local span wrapping \*/

val arrowWithTimingExecutorSideEffects = Arrow

.time(arrow.andThen(transformResultAndClassifyFailures))

.applyEffect(recordStatsAndUpdateQualityFactor)

.andThen(applyFailOpenPolicyBasedOnTransformedResult)

/\*\* Dont wrap with a span if we are not tracing \*/

tracingConfig match {

case TracingConfig.WrapWithSpanAndTracingData =>

wrapArrowWithLocalTracingSpan(arrowWithTimingExecutorSideEffects)

case TracingConfig.NoTracing =>

arrowWithTimingExecutorSideEffects

}

}

/\*\* Let-scopes a [[TraceId]] around a computation \*/

private[this] object TracingLetter extends Letter[TraceId] {

override def let[S](traceId: TraceId)(s: => S): S = Trace.letId(traceId)(s)

}

/\*\*

\* Wraps the Arrow's execution in a new trace span as a child of the current parent span

\*

\* @note Should \_\_never\_\_ be called directly!

\*

\* It's expected that the contained `arrow` will invoke [[recordTraceData]] exactly ONCE

\* during it's execution.

\*

\* @note this does not record any data about the trace, it only sets the [[Trace]] Span

\* for the execution of `arrow`

\*/

private[service] def wrapArrowWithLocalTracingSpan[In, Out](

arrow: Arrow[In, Out]

): Arrow[In, Out] =

Arrow.ifelse(

\_ => Trace.isActivelyTracing,

Arrow.let(TracingLetter)(Trace.nextId)(arrow),

arrow

)

private[this] object Tracing {

/\*\*

\* Duplicate of [[com.twitter.finagle.tracing.Tracing]]'s `localSpans` which

\* uses an un-scoped [[StatsReceiver]]

\*

\* Since we needed to roll-our-own latency measurement we are unable to increment the

\* `local\_spans` metric automatically, this is important in the event a service is

\* unexpectedly not recording spans or unexpectedly recording too many, so we manually

\* increment it

\*/

val localSpans: Counter = DefaultStatsReceiver.counter("tracing", "local\_spans")

/\*\* Local Component field of a span in the UI \*/

val localComponentTag = "lc"

val sizeTag = "product\_mixer.result.size"

val successTag = "product\_mixer.result.success"

val successValue = "success"

val cancelledTag = "product\_mixer.result.cancelled"

val failureTag = "product\_mixer.result.failure"

}

/\*\*

\* Records metadata onto the current [[Trace]] Span

\*

\* @note Should \_\_never\_\_ be called directly!

\*

\* This should be called exactly ONCE in the Arrow passed into [[wrapArrowWithLocalTracingSpan]]

\* to record data for the Span.

\*/

private[service] def recordTraceData[T](

componentStack: ComponentIdentifierStack,

componentIdentifier: ComponentIdentifier,

result: Try[T],

latency: Duration,

size: Option[Int] = None

): Unit = {

if (Trace.isActivelyTracing) {

Tracing.localSpans.incr()

val traceId = Trace.id

val endTime = Time.nowNanoPrecision

// These annotations are needed for the Zipkin UI to display the span properly

TraceServiceName().foreach(Trace.recordServiceName)

Trace.recordRpc(componentIdentifier.snakeCase) // name of the span in the UI

Trace.recordBinary(

Tracing.localComponentTag,

componentStack.peek.toString + "/" + componentIdentifier.toString)

Trace.record(Record(traceId, endTime - latency, Annotation.Message(LocalBeginAnnotation)))

Trace.record(Record(traceId, endTime, Annotation.Message(LocalEndAnnotation)))

// product mixer specific zipkin data

size.foreach(size => Trace.recordBinary(Tracing.sizeTag, size))

result match {

case Return(\_) =>

Trace.recordBinary(Tracing.successTag, Tracing.successValue)

case Throw(CancelledExceptionExtractor(e)) =>

Trace.recordBinary(Tracing.cancelledTag, e)

case Throw(e) =>

Trace.recordBinary(Tracing.failureTag, e)

}

}

}

/\*\*

\* Returns a tuple of the stats scopes for the current component and the relative scope for

\* the parent component and the current component together

\*

\* This is useful when recording stats for a component by itself as well as stats for calls to that component from it's parent.

\*

\* @example if the current component has a scope of "currentComponent" and the parent component has a scope of "parentComponent"

\* then this will return `(Seq("currentComponent"), Seq("parentComponent", "currentComponent"))`

\*/

def buildScopes(

context: Context,

currentComponentIdentifier: ComponentIdentifier

): Executor.Scopes = {

val parentScopes = context.componentStack.peek.toScopes

val componentScopes = currentComponentIdentifier.toScopes

val relativeScopes = parentScopes ++ componentScopes

Executor.Scopes(componentScopes, relativeScopes)

}

/\*\*

\* Makes a [[BroadcastStatsReceiver]] that will broadcast stats to the correct

\* current component's scope and to the scope relative to the parent.

\*/

def broadcastStatsReceiver(

context: Context,

currentComponentIdentifier: ComponentIdentifier,

statsReceiver: StatsReceiver

): StatsReceiver = {

val Executor.Scopes(componentScopes, relativeScopes) =

Executor.buildScopes(context, currentComponentIdentifier)

BroadcastStatsReceiver(

Seq(statsReceiver.scope(relativeScopes: \_\*), statsReceiver.scope(componentScopes: \_\*)))

}

/\*\*

\* Returns a feature map containing all the [[com.twitter.product\_mixer.core.feature.Feature]]s

\* stored as failures using the exception provided with as the reason wrapped in a PipelineFailure.

\* e.g, for features A & B that threw an ExampleException b, this will return:

\* { A -> Throw(PipelineFailure(...)), B -> Throw(PipelineFailure(...)) }

\*/

def featureMapWithFailuresForFeatures(

features: Set[Feature[\_, \_]],

error: Throwable,

context: Executor.Context

): FeatureMap = {

val builder = FeatureMapBuilder()

features.foreach { feature =>

val pipelineFailure = PipelineFailure(

FeatureHydrationFailed,

s"Feature hydration failed for ${feature.toString}",

Some(error),

Some(context.componentStack))

builder.addFailure(feature, pipelineFailure)

}

builder.build()

}

/\*\*

\* Validates and returns back the passed feature map if it passes validation. A feature map

\* is considered valid if it contains only the passed `registeredFeatures` features in it,

\* nothing else and nothing missing.

\*/

@throws(classOf[PipelineFailure])

def validateFeatureMap(

registeredFeatures: Set[Feature[\_, \_]],

featureMap: FeatureMap,

context: Executor.Context

): FeatureMap = {

val hydratedFeatures = featureMap.getFeatures

if (hydratedFeatures == registeredFeatures) {

featureMap

} else {

val missingFeatures = registeredFeatures -- hydratedFeatures

val unregisteredFeatures = hydratedFeatures -- registeredFeatures

throw PipelineFailure(

MisconfiguredFeatureMapFailure,

s"Unregistered features $unregisteredFeatures and missing features $missingFeatures",

None,

Some(context.componentStack)

)

}

}

object NotAMisconfiguredFeatureMapFailure {

/\*\*

\* Will return any exception that isn't a [[MisconfiguredFeatureMapFailure]] [[PipelineFailure]]

\* Allows for easy [[Arrow.handle]]ing all exceptions that aren't [[MisconfiguredFeatureMapFailure]]s

\*/

def unapply(e: Throwable): Option[Throwable] = e match {

case pipelineFailure: PipelineFailure

if pipelineFailure.category == MisconfiguredFeatureMapFailure =>

None

case e => Some(e)

}

}

/\*\*

\* contains the scopes for recording metrics for the component by itself and

\* the relative scope of that component within it's parent component scope

\*

\* @see [[Executor.buildScopes]]

\*/

case class Scopes(componentScopes: Seq[String], relativeScope: Seq[String])

/\*\*

\* Wrap the [[Throwable]] in a [[UncategorizedServerFailure]] [[PipelineFailure]] with the original

\* [[Throwable]] as the cause, even if it's already a [[PipelineFailure]].

\*

\* This ensures that any access to the stored feature will result in a meaningful [[UncategorizedServerFailure]]

\* [[com.twitter.product\_mixer.core.pipeline.pipeline\_failure.PipelineFailureCategory]] in stats which is more useful

\* for customers components which access a failed [[Feature]] than the original [[com.twitter.product\_mixer.core.pipeline.pipeline\_failure.PipelineFailureCategory]].

\*/

def uncategorizedServerFailure(

componentStack: ComponentIdentifierStack,

throwable: Throwable

): PipelineFailure = {

PipelineFailure(

UncategorizedServerFailure,

reason = "Unclassified Failure in Pipeline",

Some(throwable),

Some(componentStack)

)

}

/\*\*

\* [[PartialFunction]] that converts any [[Throwable]] into a

\* [[PipelineFailure]] based on the provided `failureClassifier`

\*/

def toPipelineFailureWithComponentIdentifierStack(

context: Context,

currentComponentIdentifier: ComponentIdentifier

): PipelineFailureClassifier = {

// if given a `currentComponentIdentifier` then ensure we correctly handle `BasedOnParentComponent` identifier types

val contextWithCurrentComponentIdentifier =

context.pushToComponentStack(currentComponentIdentifier)

PipelineFailureClassifier(

contextWithCurrentComponentIdentifier.pipelineFailureClassifier

.orElse[Throwable, PipelineFailure] {

case CancelledExceptionExtractor(throwable) => throw throwable

case pipelineFailure: PipelineFailure => pipelineFailure

case throwable =>

uncategorizedServerFailure(

contextWithCurrentComponentIdentifier.componentStack,

throwable)

}.andThen { pipelineFailure =>

pipelineFailure.componentStack match {

case \_: Some[\_] => pipelineFailure

case None =>

pipelineFailure.copy(componentStack =

Some(contextWithCurrentComponentIdentifier.componentStack))

}

}

)

}

/\*\*

\* information used by an [[Executor]] that provides context around execution

\*/

case class Context(

pipelineFailureClassifier: PipelineFailureClassifier,

componentStack: ComponentIdentifierStack) {

def pushToComponentStack(newComponentIdentifier: ComponentIdentifier): Context =

copy(componentStack = componentStack.push(newComponentIdentifier))

}

}