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

import com.twitter.finagle.mtls.authentication.ServiceIdentifier

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.tracing.Trace

import com.twitter.finagle.transport.Transport

import com.twitter.product\_mixer.core.functional\_component.common.access\_policy.AccessPolicy

import com.twitter.product\_mixer.core.functional\_component.common.alert.Alert

import com.twitter.product\_mixer.core.functional\_component.gate.Gate

import com.twitter.product\_mixer.core.gate.DenyLoggedOutUsersGate

import com.twitter.product\_mixer.core.gate.ParamGate

import com.twitter.product\_mixer.core.gate.ParamGate.EnabledGateSuffix

import com.twitter.product\_mixer.core.gate.ParamGate.SupportedClientGateSuffix

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

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.model.marshalling.request.Request

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

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

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

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

import com.twitter.product\_mixer.core.pipeline.mixer.MixerPipelineBuilderFactory

import com.twitter.product\_mixer.core.pipeline.mixer.MixerPipelineConfig

import com.twitter.product\_mixer.core.pipeline.mixer.MixerPipelineResult

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

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

import com.twitter.product\_mixer.core.pipeline.recommendation.RecommendationPipelineBuilderFactory

import com.twitter.product\_mixer.core.pipeline.recommendation.RecommendationPipelineConfig

import com.twitter.product\_mixer.core.pipeline.recommendation.RecommendationPipelineResult

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

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

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

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

import com.twitter.product\_mixer.core.service.gate\_executor.GateExecutor

import com.twitter.product\_mixer.core.service.gate\_executor.GateExecutorResult

import com.twitter.product\_mixer.core.service.gate\_executor.StoppedGateException

import com.twitter.product\_mixer.core.service.pipeline\_execution\_logger.PipelineExecutionLogger

import com.twitter.product\_mixer.core.service.pipeline\_executor.PipelineExecutor

import com.twitter.product\_mixer.core.service.pipeline\_executor.PipelineExecutorRequest

import com.twitter.product\_mixer.core.service.pipeline\_executor.PipelineExecutorResult

import com.twitter.product\_mixer.core.service.pipeline\_selector\_executor.PipelineSelectorExecutor

import com.twitter.product\_mixer.core.service.pipeline\_selector\_executor.PipelineSelectorExecutorResult

import com.twitter.product\_mixer.core.service.quality\_factor\_executor.QualityFactorExecutorResult

import com.twitter.stitch.Arrow

import com.twitter.stringcenter.client.StringCenterRequestContext

import com.twitter.stringcenter.client.stitch.StringCenterRequestContextLetter

import com.twitter.timelines.configapi.Params

import com.twitter.util.logging.Logging

import org.slf4j.MDC

class ProductPipelineBuilder[TRequest <: Request, Query <: PipelineQuery, Response](

gateExecutor: GateExecutor,

pipelineSelectorExecutor: PipelineSelectorExecutor,

pipelineExecutor: PipelineExecutor,

mixerPipelineBuilderFactory: MixerPipelineBuilderFactory,

recommendationPipelineBuilderFactory: RecommendationPipelineBuilderFactory,

override val statsReceiver: StatsReceiver,

pipelineExecutionLogger: PipelineExecutionLogger)

extends PipelineBuilder[ProductPipelineRequest[TRequest]]

with Logging { builder =>

override type UnderlyingResultType = Response

override type PipelineResultType = ProductPipelineResult[Response]

/\*\*

\* Query Transformer Step is implemented inline instead of using an executor.

\*

\* It's a simple, synchronous step that executes the query transformer.

\*

\* Since the output of the transformer is used in multiple other steps (Gate, Pipeline Execution),

\* we've promoted the transformer to a step so that it's outputs can be reused easily.

\*/

def pipelineQueryTransformerStep(

queryTransformer: (TRequest, Params) => Query,

context: Executor.Context

): Step[ProductPipelineRequest[TRequest], Query] =

new Step[ProductPipelineRequest[TRequest], Query] {

override def identifier: PipelineStepIdentifier =

ProductPipelineConfig.pipelineQueryTransformerStep

override def executorArrow: Arrow[ProductPipelineRequest[TRequest], Query] = {

wrapWithErrorHandling(context, identifier)(

Arrow.map[ProductPipelineRequest[TRequest], Query] {

case ProductPipelineRequest(request, params) => queryTransformer(request, params)

}

)

}

override def inputAdaptor(

query: ProductPipelineRequest[TRequest],

previousResult: ProductPipelineResult[Response]

): ProductPipelineRequest[TRequest] = query

override def resultUpdater(

previousPipelineResult: ProductPipelineResult[Response],

executorResult: Query

): ProductPipelineResult[Response] =

previousPipelineResult.copy(transformedQuery = Some(executorResult))

}

def qualityFactorStep(

qualityFactorStatus: QualityFactorStatus

): Step[Query, QualityFactorExecutorResult] = {

new Step[Query, QualityFactorExecutorResult] {

override def identifier: PipelineStepIdentifier = ProductPipelineConfig.qualityFactorStep

override def executorArrow: Arrow[Query, QualityFactorExecutorResult] =

Arrow

.map[Query, QualityFactorExecutorResult] { \_ =>

QualityFactorExecutorResult(

pipelineQualityFactors =

qualityFactorStatus.qualityFactorByPipeline.mapValues(\_.currentValue)

)

}

override def inputAdaptor(

query: ProductPipelineRequest[TRequest],

previousResult: ProductPipelineResult[Response]

): Query = previousResult.transformedQuery

.getOrElse {

throw InvalidStepStateException(identifier, "TransformedQuery")

}.asInstanceOf[Query]

override def resultUpdater(

previousPipelineResult: ProductPipelineResult[Response],

executorResult: QualityFactorExecutorResult

): ProductPipelineResult[Response] = {

previousPipelineResult.copy(

transformedQuery = previousPipelineResult.transformedQuery.map {

case queryWithQualityFactor: HasQualityFactorStatus =>

queryWithQualityFactor

.withQualityFactorStatus(qualityFactorStatus).asInstanceOf[Query]

case query =>

query

},

qualityFactorResult = Some(executorResult)

)

}

}

}

def gatesStep(

gates: Seq[Gate[Query]],

context: Executor.Context

): Step[Query, GateExecutorResult] = new Step[Query, GateExecutorResult] {

override def identifier: PipelineStepIdentifier = ProductPipelineConfig.gatesStep

override def executorArrow: Arrow[Query, GateExecutorResult] = {

gateExecutor.arrow(gates, context)

}

override def inputAdaptor(

query: ProductPipelineRequest[TRequest],

previousResult: ProductPipelineResult[Response]

): Query = previousResult.transformedQuery

.getOrElse {

throw InvalidStepStateException(identifier, "TransformedQuery")

}.asInstanceOf[Query]

override def resultUpdater(

previousPipelineResult: ProductPipelineResult[Response],

executorResult: GateExecutorResult

): ProductPipelineResult[Response] =

previousPipelineResult.copy(gateResult = Some(executorResult))

}

def pipelineSelectorStep(

pipelineByIdentifer: Map[ComponentIdentifier, Pipeline[Query, Response]],

pipelineSelector: Query => ComponentIdentifier,

context: Executor.Context

): Step[Query, PipelineSelectorExecutorResult] =

new Step[Query, PipelineSelectorExecutorResult] {

override def identifier: PipelineStepIdentifier = ProductPipelineConfig.pipelineSelectorStep

override def executorArrow: Arrow[

Query,

PipelineSelectorExecutorResult

] = pipelineSelectorExecutor.arrow(pipelineByIdentifer, pipelineSelector, context)

override def inputAdaptor(

query: ProductPipelineRequest[TRequest],

previousResult: ProductPipelineResult[Response]

): Query =

previousResult.transformedQuery

.getOrElse(throw InvalidStepStateException(identifier, "TransformedQuery")).asInstanceOf[

Query]

override def resultUpdater(

previousPipelineResult: ProductPipelineResult[Response],

executorResult: PipelineSelectorExecutorResult

): ProductPipelineResult[Response] =

previousPipelineResult.copy(pipelineSelectorResult = Some(executorResult))

}

def pipelineExecutionStep(

pipelineByIdentifier: Map[ComponentIdentifier, Pipeline[Query, Response]],

qualityFactorObserverByPipeline: Map[ComponentIdentifier, QualityFactorObserver],

context: Executor.Context

): Step[PipelineExecutorRequest[Query], PipelineExecutorResult[Response]] =

new Step[PipelineExecutorRequest[Query], PipelineExecutorResult[Response]] {

override def identifier: PipelineStepIdentifier = ProductPipelineConfig.pipelineExecutionStep

override def executorArrow: Arrow[

PipelineExecutorRequest[Query],

PipelineExecutorResult[Response]

] = {

pipelineExecutor.arrow(pipelineByIdentifier, qualityFactorObserverByPipeline, context)

}

override def inputAdaptor(

request: ProductPipelineRequest[TRequest],

previousResult: ProductPipelineResult[Response]

): PipelineExecutorRequest[Query] = {

val query = previousResult.transformedQuery

.getOrElse {

throw InvalidStepStateException(identifier, "TransformedQuery")

}.asInstanceOf[Query]

val pipelineIdentifier = previousResult.pipelineSelectorResult

.map(\_.pipelineIdentifier).getOrElse {

throw InvalidStepStateException(identifier, "PipelineSelectorResult")

}

PipelineExecutorRequest(query, pipelineIdentifier)

}

override def resultUpdater(

previousPipelineResult: ProductPipelineResult[Response],

executorResult: PipelineExecutorResult[Response]

): ProductPipelineResult[Response] = {

val mixerPipelineResult = executorResult.pipelineResult match {

case mixerPipelineResult: MixerPipelineResult[Response] @unchecked =>

Some(mixerPipelineResult)

case \_ =>

None

}

val recommendationPipelineResult = executorResult.pipelineResult match {

case recommendationPipelineResult: RecommendationPipelineResult[

\_,

Response

] @unchecked =>

Some(recommendationPipelineResult)

case \_ =>

None

}

previousPipelineResult.copy(

mixerPipelineResult = mixerPipelineResult,

recommendationPipelineResult = recommendationPipelineResult,

traceId = Trace.idOption.map(\_.traceId.toString()),

result = executorResult.pipelineResult.result

)

}

}

def build(

parentComponentIdentifierStack: ComponentIdentifierStack,

config: ProductPipelineConfig[TRequest, Query, Response]

): ProductPipeline[TRequest, Response] = {

val pipelineIdentifier = config.identifier

val context = Executor.Context(

PipelineFailureClassifier(

config.failureClassifier.orElse(StoppedGateException.classifier(ProductDisabled))),

parentComponentIdentifierStack.push(pipelineIdentifier)

)

val denyLoggedOutUsersGate = if (config.denyLoggedOutUsers) {

Some(DenyLoggedOutUsersGate(pipelineIdentifier))

} else {

None

}

val enabledGate: ParamGate =

ParamGate(pipelineIdentifier + EnabledGateSuffix, config.paramConfig.EnabledDeciderParam)

val supportedClientGate =

ParamGate(

pipelineIdentifier + SupportedClientGateSuffix,

config.paramConfig.SupportedClientParam)

/\*\*

\* Evaluate enabled decider gate first since if it's off, there is no reason to proceed

\* Next evaluate supported client feature switch gate, followed by customer configured gates

\*/

val allGates =

denyLoggedOutUsersGate.toSeq ++: enabledGate +: supportedClientGate +: config.gates

val childPipelines: Seq[Pipeline[Query, Response]] =

config.pipelines.map {

case mixerConfig: MixerPipelineConfig[Query, \_, Response] =>

mixerConfig.build(context.componentStack, mixerPipelineBuilderFactory)

case recommendationConfig: RecommendationPipelineConfig[Query, \_, \_, Response] =>

recommendationConfig.build(context.componentStack, recommendationPipelineBuilderFactory)

case other =>

throw new IllegalArgumentException(

s"Product Pipelines only support Mixer and Recommendation pipelines, not $other")

}

val pipelineByIdentifier: Map[ComponentIdentifier, Pipeline[Query, Response]] =

childPipelines.map { pipeline =>

(pipeline.identifier, pipeline)

}.toMap

val qualityFactorStatus: QualityFactorStatus =

QualityFactorStatus.build(config.qualityFactorConfigs)

val qualityFactorObserverByPipeline = qualityFactorStatus.qualityFactorByPipeline.mapValues {

qualityFactor =>

qualityFactor.buildObserver()

}

buildGaugesForQualityFactor(pipelineIdentifier, qualityFactorStatus, statsReceiver)

/\*\*

\* Initialize MDC with access logging with everything we have at request time. We can put

\* more stuff into MDC later down the pipeline, but at risk of exceptions/errors preventing

\* them from being added

\*/

val mdcInitArrow =

Arrow.map[ProductPipelineRequest[TRequest], ProductPipelineRequest[TRequest]] { request =>

val serviceIdentifier = ServiceIdentifier.fromCertificate(Transport.peerCertificate)

MDC.put("product", config.product.identifier.name)

MDC.put("serviceIdentifier", ServiceIdentifier.asString(serviceIdentifier))

request

}

val builtSteps = Seq(

pipelineQueryTransformerStep(config.pipelineQueryTransformer, context),

qualityFactorStep(qualityFactorStatus),

gatesStep(allGates, context),

pipelineSelectorStep(pipelineByIdentifier, config.pipelineSelector, context),

pipelineExecutionStep(pipelineByIdentifier, qualityFactorObserverByPipeline, context)

)

val underlying: Arrow[ProductPipelineRequest[TRequest], ProductPipelineResult[Response]] =

buildCombinedArrowFromSteps(

steps = builtSteps,

context = context,

initialEmptyResult = ProductPipelineResult.empty,

stepsInOrderFromConfig = ProductPipelineConfig.stepsInOrder

)

/\*\*

\* Unlike other components and pipelines, [[ProductPipeline]] must be observed in the

\* [[ProductPipelineBuilder]] directly because the resulting [[ProductPipeline.arrow]]

\* is run directly without an executor so must contain all stats.

\*/

val observed =

wrapProductPipelineWithExecutorBookkeeping[

ProductPipelineRequest[TRequest],

ProductPipelineResult[Response]

](context, pipelineIdentifier)(underlying)

val finalArrow: Arrow[ProductPipelineRequest[TRequest], ProductPipelineResult[Response]] =

Arrow

.letWithArg[

ProductPipelineRequest[TRequest],

ProductPipelineResult[Response],

StringCenterRequestContext](StringCenterRequestContextLetter)(request =>

StringCenterRequestContext(

request.request.clientContext.languageCode,

request.request.clientContext.countryCode

))(

mdcInitArrow

.andThen(observed)

.onSuccess(result => result.transformedQuery.map(pipelineExecutionLogger(\_, result))))

val configFromBuilder = config

new ProductPipeline[TRequest, Response] {

override private[core] val config: ProductPipelineConfig[TRequest, \_, Response] =

configFromBuilder

override val arrow: Arrow[ProductPipelineRequest[TRequest], ProductPipelineResult[Response]] =

finalArrow

override val identifier: ProductPipelineIdentifier = pipelineIdentifier

override val alerts: Seq[Alert] = config.alerts

override val debugAccessPolicies: Set[AccessPolicy] = config.debugAccessPolicies

override val children: Seq[Component] = allGates ++ childPipelines

}

}

}