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

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

import com.twitter.product\_mixer.core.functional\_component.feature\_hydrator.QueryFeatureHydrator

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

import com.twitter.product\_mixer.core.functional\_component.premarshaller.DomainMarshaller

import com.twitter.product\_mixer.core.functional\_component.selector.Selector

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

import com.twitter.product\_mixer.core.functional\_component.marshaller.TransportMarshaller

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

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

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

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

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

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

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

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

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

import com.twitter.product\_mixer.core.pipeline.candidate.CandidatePipelineConfig

import com.twitter.product\_mixer.core.pipeline.candidate.DependentCandidatePipelineConfig

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

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

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

/\*\*

\* This is the configuration necessary to generate a Mixer Pipeline. Product code should create a

\* MixerPipelineConfig, and then use a MixerPipelineBuilder to get the final MixerPipeline which can

\* process requests.

\*

\* @tparam Query - The domain model for the query or request

\* @tparam UnmarshalledResultType - The result type of the pipeline, but before marshalling to a wire protocol like URT

\* @tparam Result - The final result that will be served to users

\*/

trait MixerPipelineConfig[Query <: PipelineQuery, UnmarshalledResultType <: HasMarshalling, Result]

extends PipelineConfig {

override val identifier: MixerPipelineIdentifier

/\*\*

\* Mixer Pipeline Gates will be executed before any other step (including retrieval from candidate

\* pipelines). They're executed sequentially, and any "Stop" result will prevent pipeline execution.

\*/

def gates: Seq[Gate[Query]] = Seq.empty

/\*\*

\* A mixer pipeline can fetch query-level features before candidate pipelines are executed.

\*/

def fetchQueryFeatures: Seq[QueryFeatureHydrator[Query]] = Seq.empty

/\*\*

\* For query-level features that are dependent on query-level features from [[fetchQueryFeatures]]

\*/

def fetchQueryFeaturesPhase2: Seq[QueryFeatureHydrator[Query]] = Seq.empty

/\*\*

\* Candidate pipelines retrieve candidates for possible inclusion in the result

\*/

def candidatePipelines: Seq[CandidatePipelineConfig[Query, \_, \_, \_]]

/\*\*

\* Dependent candidate pipelines to retrieve candidates that depend on the result of [[candidatePipelines]]

\* [[DependentCandidatePipelineConfig]] have access to the list of previously retrieved & decorated

\* candidates for use in constructing the query object.

\*/

def dependentCandidatePipelines: Seq[DependentCandidatePipelineConfig[Query, \_, \_, \_]] = Seq.empty

/\*\*

\* [[defaultFailOpenPolicy]] is the [[FailOpenPolicy]] that will be applied to any candidate

\* pipeline that isn't in the [[failOpenPolicies]] map. By default Candidate Pipelines will fail

\* open for Closed Gates only.

\*/

def defaultFailOpenPolicy: FailOpenPolicy = FailOpenPolicy(Set(ClosedGate))

/\*\*

\* [[failOpenPolicies]] associates [[FailOpenPolicy]]s to specific candidate pipelines using

\* [[CandidatePipelineIdentifier]].

\*

\* @note these [[FailOpenPolicy]]s override the [[defaultFailOpenPolicy]] for a mapped

\* Candidate Pipeline.

\*/

def failOpenPolicies: Map[CandidatePipelineIdentifier, FailOpenPolicy] = Map.empty

/\*\*

\*\* [[qualityFactorConfigs]] associates [[QualityFactorConfig]]s to specific candidate pipelines

\* using [[CandidatePipelineIdentifier]].

\*/

def qualityFactorConfigs: Map[CandidatePipelineIdentifier, QualityFactorConfig] =

Map.empty

/\*\*

\* Selectors are executed in sequential order to combine the candidates into a result

\*/

def resultSelectors: Seq[Selector[Query]]

/\*\*

\* Mixer result side effects that are executed after selection and domain marshalling

\*/

def resultSideEffects: Seq[PipelineResultSideEffect[Query, UnmarshalledResultType]] = Seq()

/\*\*

\* Domain marshaller transforms the selections into the model expected by the marshaller

\*/

def domainMarshaller: DomainMarshaller[Query, UnmarshalledResultType]

/\*\*

\* Transport marshaller transforms the model into our line-level API like URT or JSON

\*/

def transportMarshaller: TransportMarshaller[UnmarshalledResultType, Result]

/\*\*

\* A pipeline can define a partial function to rescue failures here. They will be treated as failures

\* from a monitoring standpoint, and cancellation exceptions will always be propagated (they cannot be caught here).

\*/

def failureClassifier: PartialFunction[Throwable, PipelineFailure] = PartialFunction.empty

/\*\*

\* Alert can be used to indicate the pipeline's service level objectives. Alerts and

\* dashboards will be automatically created based on this information.

\*/

val alerts: Seq[Alert] = Seq.empty

/\*\*

\* This method is used by the product mixer framework to build the pipeline.

\*/

private[core] final def build(

parentComponentIdentifierStack: ComponentIdentifierStack,

builder: MixerPipelineBuilderFactory

): MixerPipeline[Query, Result] =

builder.get.build(parentComponentIdentifierStack, this)

}

object MixerPipelineConfig extends PipelineConfigCompanion {

val qualityFactorStep: PipelineStepIdentifier = PipelineStepIdentifier("QualityFactor")

val gatesStep: PipelineStepIdentifier = PipelineStepIdentifier("Gates")

val fetchQueryFeaturesStep: PipelineStepIdentifier = PipelineStepIdentifier("FetchQueryFeatures")

val fetchQueryFeaturesPhase2Step: PipelineStepIdentifier =

PipelineStepIdentifier("FetchQueryFeaturesPhase2")

val candidatePipelinesStep: PipelineStepIdentifier = PipelineStepIdentifier("CandidatePipelines")

val dependentCandidatePipelinesStep: PipelineStepIdentifier =

PipelineStepIdentifier("DependentCandidatePipelines")

val resultSelectorsStep: PipelineStepIdentifier = PipelineStepIdentifier("ResultSelectors")

val domainMarshallerStep: PipelineStepIdentifier = PipelineStepIdentifier("DomainMarshaller")

val resultSideEffectsStep: PipelineStepIdentifier = PipelineStepIdentifier("ResultSideEffects")

val transportMarshallerStep: PipelineStepIdentifier = PipelineStepIdentifier(

"TransportMarshaller")

/\*\* All the Steps which are executed by a [[MixerPipeline]] in the order in which they are run \*/

override val stepsInOrder: Seq[PipelineStepIdentifier] = Seq(

qualityFactorStep,

gatesStep,

fetchQueryFeaturesStep,

fetchQueryFeaturesPhase2Step,

asyncFeaturesStep(candidatePipelinesStep),

candidatePipelinesStep,

asyncFeaturesStep(dependentCandidatePipelinesStep),

dependentCandidatePipelinesStep,

asyncFeaturesStep(resultSelectorsStep),

resultSelectorsStep,

domainMarshallerStep,

asyncFeaturesStep(resultSideEffectsStep),

resultSideEffectsStep,

transportMarshallerStep

)

/\*\*

\* All the Steps which an [[com.twitter.product\_mixer.core.functional\_component.feature\_hydrator.AsyncHydrator AsyncHydrator]]

\* can be configured to [[com.twitter.product\_mixer.core.functional\_component.feature\_hydrator.AsyncHydrator.hydrateBefore hydrateBefore]]

\*/

override val stepsAsyncFeatureHydrationCanBeCompletedBy: Set[PipelineStepIdentifier] = Set(

candidatePipelinesStep,

dependentCandidatePipelinesStep,

resultSelectorsStep,

resultSideEffectsStep

)

}