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

import com.twitter.finagle.stats.StatsReceiver

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

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

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

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

import com.twitter.product\_mixer.core.model.marshalling.request.Product

import com.twitter.product\_mixer.core.model.marshalling.request.Request

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

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

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

import com.twitter.product\_mixer.core.service.component\_registry.ComponentRegistry

import com.twitter.product\_mixer.core.service.component\_registry.ComponentRegistrySnapshot

import com.twitter.product\_mixer.shared\_library.observer.Observer

import com.twitter.util.Try

import com.twitter.util.Var

import com.twitter.util.logging.Logging

import javax.inject.Inject

import javax.inject.Singleton

import scala.reflect.runtime.universe.\_

@Singleton

class ProductPipelineRegistry @Inject() (

componentRegistry: ComponentRegistry,

productPipelineRegistryConfig: ProductPipelineRegistryConfig,

productPipelineBuilderFactory: ProductPipelineBuilderFactory,

statsReceiver: StatsReceiver)

extends Logging {

private val rootIdentifierStack = ComponentIdentifierStack(RootIdentifier())

private val rebuildObserver =

Observer.function[Unit](statsReceiver, "ProductPipelineRegistry", "rebuild")

/\*\*

\* Internal state of ProductPipelineRegistry.

\*

\* Build once on startup, and later whenever `rebuild()` is called.

\*/

private[this] val productPipelineByProduct =

Var[Map[Product, ProductPipeline[\_ <: Request, \_]]](buildProductPipelineByProduct())

/\*\*

\* Triggers a rebuild of the ProductPipelineRegistry and also the ComponentRegistry

\*

\* Failed rebuilds will throw an exception - likely one of the listed ones - and the product

\* registry and component registry will not be modified.

\*

\* @throws MultipleProductPipelinesForAProductException

\* @throws ComponentIdentifierCollisionException

\* @throws ChildComponentCollisionException

\*/

private[core] def rebuild(): Unit = {

Try {

rebuildObserver {

productPipelineByProduct.update(buildProductPipelineByProduct())

}

}.onFailure { ex =>

error("Failed to rebuild ProductPipelineRegistry", ex)

}.get()

}

/\*\*

\* register the provided pipeline recursively register all of it's children components

\* that are added to the [[Pipeline]]'s [[Pipeline.children]]

\*/

private def registerPipelineAndChildren(

componentRegistrySnapshot: ComponentRegistrySnapshot,

pipeline: Pipeline[\_, \_],

parentIdentifierStack: ComponentIdentifierStack

): Unit = {

val identifierStackString =

s"${parentIdentifierStack.componentIdentifiers.reverse.mkString("\t->\t")}\t->\t${pipeline.identifier}"

info(identifierStackString)

componentRegistrySnapshot.register(

component = pipeline,

parentIdentifierStack = parentIdentifierStack)

val identifierStackWithCurrentPipeline = parentIdentifierStack.push(pipeline.identifier)

pipeline.children.foreach {

case childPipeline: Pipeline[\_, \_] =>

info(s"$identifierStackString\t->\t${childPipeline.identifier}")

registerPipelineAndChildren(

componentRegistrySnapshot,

childPipeline,

identifierStackWithCurrentPipeline)

case component =>

info(s"$identifierStackString\t->\t${component.identifier}")

componentRegistrySnapshot.register(

component = component,

parentIdentifierStack = identifierStackWithCurrentPipeline)

}

}

/\*

\* Internal method (not for callers outside of this class, see rebuild() for those)

\*

\* Produces an updated Map[Product, ProductPipeline] and also refreshes the global component registry

\*/

private[this] def buildProductPipelineByProduct(

): Map[Product, ProductPipeline[\_ <: Request, \_]] = {

// Build a new component registry snapshot.

val newComponentRegistry = new ComponentRegistrySnapshot()

info(

"Registering all products, pipelines, and components (this may be helpful if you encounter dependency injection errors)")

info("debug details are in the form of `parent -> child`")

// handle the case of multiple ProductPipelines having the same product

checkForAndThrowMultipleProductPipelinesForAProduct()

// Build a Map[Product, ProductPipeline], registering everything in the new component registry recursively

val pipelinesByProduct: Map[Product, ProductPipeline[\_ <: Request, \_]] =

productPipelineRegistryConfig.productPipelineConfigs.map { productPipelineConfig =>

val product = productPipelineConfig.product

info(s"Recursively registering ${product.identifier}")

// gets the ComponentIdentifierStack without the RootIdentifier since

// we don't want RootIdentifier to show up in stats or errors

val productPipeline =

productPipelineBuilderFactory.get.build(

ComponentIdentifierStack(product.identifier),

productPipelineConfig)

// gets RootIdentifier so we can register Products under the correct hierarchy

newComponentRegistry.register(product, rootIdentifierStack)

registerPipelineAndChildren(

newComponentRegistry,

productPipeline,

rootIdentifierStack.push(product.identifier))

// In addition to registering the component in the main registry, we want to maintain a map of

// product to the product pipeline to allow for O(1) lookup by product on the request hot path

product -> productPipeline

}.toMap

info(

s"Successfully registered ${newComponentRegistry.getAllRegisteredComponents

.count(\_.identifier.isInstanceOf[ProductIdentifier])} products and " +

s"${newComponentRegistry.getAllRegisteredComponents.length} " +

s"components total, query the component registry endpoint for details")

componentRegistry.set(newComponentRegistry)

pipelinesByProduct

}

// handle the case of multiple ProductPipelines having the same product

private def checkForAndThrowMultipleProductPipelinesForAProduct(): Unit = {

productPipelineRegistryConfig.productPipelineConfigs.groupBy(\_.product.identifier).foreach {

case (product, productPipelines) if productPipelines.length != 1 =>

throw new MultipleProductPipelinesForAProductException(

product,

productPipelines.map(\_.identifier))

case \_ =>

}

}

def getProductPipeline[MixerRequest <: Request: TypeTag, ResponseType: TypeTag](

product: Product

): ProductPipeline[MixerRequest, ResponseType] = {

// Check and cast the bounded existential types to the concrete types

(typeOf[MixerRequest], typeOf[ResponseType]) match {

case (req, res) if req =:= typeOf[MixerRequest] && res =:= typeOf[ResponseType] =>

productPipelineByProduct.sample

.getOrElse(product, throw new ProductNotFoundException(product))

.asInstanceOf[ProductPipeline[MixerRequest, ResponseType]]

case \_ =>

throw new UnknownPipelineResponseException(product)

}

}

}

class ProductNotFoundException(product: Product)

extends RuntimeException(s"No Product found for $product")

class UnknownPipelineResponseException(product: Product)

extends RuntimeException(s"Unknown pipeline response for $product")

class MultipleProductPipelinesForAProductException(

product: ProductIdentifier,

pipelineIdentifiers: Seq[ProductPipelineIdentifier])

extends IllegalStateException(s"Multiple ProductPipelines found for $product, found " +

s"${pipelineIdentifiers

.map(productPipelineIdentifier => s"$productPipelineIdentifier from ${productPipelineIdentifier.file}")

.mkString(", ")} ")