package com.twitter.product\_mixer.component\_library.experiments.metrics

import com.twitter.product\_mixer.component\_library.experiments.metrics.PlaceholderConfig.PlaceholdersMap

import reflect.ClassTag

import scala.reflect.runtime.universe.\_

import scala.util.matching.Regex

case class MatchedPlaceholder(outerKey: String, innerKey: Option[String] = None)

object MetricTemplates {

// Matches "${placeholder}" where `placeholder` is in a matched group

val PlaceholderPattern: Regex = "\\$\\{([^\\}]+)\\}".r.unanchored

// Matches "${placeholder[index]}" where `placeholder` and `index` are in different matched groups

val IndexedPlaceholderPattern: Regex = "\\$\\{([^\\[]+)\\[([^\\]]+)\\]\\}".r.unanchored

val DefaultFieldName = "name"

def interpolate(inputTemplate: String, placeholders: PlaceholdersMap): Seq[String] = {

val matchedPlaceholders = getMatchedPlaceholders(inputTemplate)

val groupedPlaceholders = matchedPlaceholders.groupBy(\_.outerKey)

val placeholderKeyValues = getPlaceholderKeyValues(groupedPlaceholders, placeholders)

val (keys, values) = (placeholderKeyValues.map(\_.\_1), placeholderKeyValues.map(\_.\_2))

val cross: Seq[List[Named]] = crossProduct(values)

val mirror = runtimeMirror(getClass.getClassLoader) // necessary for reflection

for {

interpolatables <- cross

} yield {

assert(

keys.length == interpolatables.length,

s"Unexpected length mismatch between $keys and $interpolatables")

var replacementStr = inputTemplate

keys.zip(interpolatables).foreach {

case (key, interpolatable) =>

val accessors = caseAccessors(mirror, interpolatable)

groupedPlaceholders(key).foreach { placeholder: MatchedPlaceholder =>

val templateKey = generateTemplateKey(placeholder)

val fieldName = placeholder.innerKey.getOrElse(DefaultFieldName)

val fieldValue = getFieldValue(mirror, interpolatable, accessors, fieldName)

replacementStr = replacementStr.replaceAll(templateKey, fieldValue)

}

}

replacementStr

}

}

def getMatchedPlaceholders(inputTemplate: String): Seq[MatchedPlaceholder] = {

for {

matched <- PlaceholderPattern.findAllIn(inputTemplate).toSeq

} yield {

val matchedWithIndexOpt = IndexedPlaceholderPattern.findFirstMatchIn(matched)

val (outer, inner) = matchedWithIndexOpt

.map { matchedWithIndex =>

(matchedWithIndex.group(1), Some(matchedWithIndex.group(2)))

}.getOrElse((matched, None))

val outerKey = unwrap(outer)

val innerKey = inner.map(unwrap(\_))

MatchedPlaceholder(outerKey, innerKey)

}

}

def unwrap(str: String): String =

str.stripPrefix("${").stripSuffix("}")

def wrap(str: String): String =

"\\$\\{" + str + "\\}"

def getPlaceholderKeyValues(

groupedPlaceholders: Map[String, Seq[MatchedPlaceholder]],

placeholders: PlaceholdersMap

): Seq[(String, Seq[Named])] = {

groupedPlaceholders.toSeq

.map {

case (outerKey, \_) =>

val placeholderValues = placeholders.getOrElse(

outerKey,

throw new RuntimeException(s"Failed to find values of $outerKey in placeholders"))

outerKey -> placeholderValues

}

}

def crossProduct[T](seqOfSeqOfItems: Seq[Seq[T]]): Seq[List[T]] = {

if (seqOfSeqOfItems.isEmpty) {

List(Nil)

} else {

for {

// for every item in the head list

item <- seqOfSeqOfItems.head

// for every result (List) based on the cross-product of tail

resultList <- crossProduct(seqOfSeqOfItems.tail)

} yield {

item :: resultList

}

}

}

def generateTemplateKey(matched: MatchedPlaceholder): String = {

matched.innerKey match {

case None => wrap(matched.outerKey)

case Some(innerKeyString) => wrap(matched.outerKey + "\\[" + innerKeyString + "\\]")

}

}

// Given an instance and a field name, use reflection to get its value.

def getFieldValue[T: ClassTag](

mirror: Mirror,

cls: T,

accessors: Map[String, MethodSymbol],

fieldName: String

): String = {

val instance: InstanceMirror = mirror.reflect(cls)

val accessor = accessors.getOrElse(

fieldName,

throw new RuntimeException(s"Failed to find value of $fieldName for $cls"))

instance.reflectField(accessor).get.toString // .get is safe due to check above

}

// Given an instance, use reflection to get a mapping for field name -> symbol

def caseAccessors[T: ClassTag](mirror: Mirror, cls: T): Map[String, MethodSymbol] = {

val classSymbol = mirror.classSymbol(cls.getClass)

classSymbol.toType.members.collect {

case m: MethodSymbol if m.isCaseAccessor => (m.name.toString -> m)

}.toMap

}

}