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

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

import java.io.File

import java.io.PrintWriter

import scala.collection.immutable.ListSet

import scala.io.Source

import scopt.OptionParser

private case class MetricTemplateCLIConfig(

// default values required for OptionParser

templateFileName: String = null,

outputFileName: String = null,

metricGroupName: String = null,

metricGroupDesc: String = null,

metricGroupId: Option[Long] = None,

absolutePath: Option[String] = None)

trait MetricTemplateCLIRunner {

def templateDir: String

def placeholders: PlaceholdersMap

private val ProgramName = "Metric Template CLI"

private val VersionNumber = "1.0"

private def mkPath(fileName: String, absolutePath: Option[String]): String = {

val relativeDir = s"$templateDir/$fileName"

absolutePath match {

case Some(path) => s"$path/$relativeDir"

case \_ => relativeDir

}

}

def main(args: Array[String]): Unit = {

val parser = new OptionParser[MetricTemplateCLIConfig](ProgramName) {

head(ProgramName, VersionNumber)

// option invoked by -o or --output

opt[String]('o', "output")

.required()

.valueName("<file>")

.action((value, config) => config.copy(outputFileName = value))

.text("output CSV file with interpolated lines")

// option invoked by -t or --template

opt[String]('t', "template")

.required()

.valueName("<file>")

.action((value, config) => config.copy(templateFileName = value))

.text(

s"input template file (see README.md for template format). Path is relative to $templateDir.")

// option invoked by -n or --name

opt[String]('n', "name")

.required()

.valueName("<groupName>")

.action((value, config) => config.copy(metricGroupName = value))

.text("metric group name")

// option invoked by -d or --description

opt[String]('d', "description")

.required()

.valueName("<groupDescription>")

.action((value, config) => config.copy(metricGroupDesc = value))

.text("metric group description")

// option invoked by --id

opt[Long]("id")

.optional()

.valueName("<groupId>")

.action((value, config) => config.copy(metricGroupId = Some(value)))

.text("metric group ID (metric MUST be created in go/ddg)")

// option invoked by -p or --path

opt[String]('p', "path")

.optional()

.valueName("<directory>")

.action((value, config) => config.copy(absolutePath = Some(value)))

.text(s"absolute path pointing to the $templateDir. Required by bazel")

}

parser.parse(args, MetricTemplateCLIConfig()) match {

case Some(config) =>

val templateLines =

Source.fromFile(mkPath(config.templateFileName, config.absolutePath)).getLines.toList

val interpolatedLines = templateLines

.filter(!\_.startsWith("#")).flatMap(MetricTemplates.interpolate(\_, placeholders))

val writer = new PrintWriter(new File(mkPath(config.outputFileName, config.absolutePath)))

val metrics = interpolatedLines.map(Metric.fromLine)

println(s"${metrics.size} metric definitions found in template file.")

val dupMetrics = metrics.groupBy(identity).collect {

case (dup, lst) if lst.lengthCompare(1) > 0 => dup

}

println(s"\nWARNING: ${dupMetrics.size} Duplicate metric definition(s)\n$dupMetrics\n")

val metricGroup = MetricGroup(

config.metricGroupId,

config.metricGroupName,

config.metricGroupDesc,

metrics.to[ListSet])

println(s"${metricGroup.uniqueMetricNames.size} unique DDG metrics with " +

s"${metricGroup.metrics.size} metric definitions in '${metricGroup.name}' metric group.")

writer.write(metricGroup.toCsv)

writer.close()

case \_ =>

// arguments are bad, error message will have been displayed

}

}

}