GET CODE FROM GITHUB REPOSITORY

Jenkinsfile (Scripted Pipeline for Adire E-commers)

Node {

Stage (‘Code Get from GitHub’) {

//

Stage (‘Build’) {

//

}

Stage (‘test’) {

//

}

Stage (‘deploy to Tomcat’) {

//

}

}

Commit to tomcat

// Copyright (c) IBM 2017,2018

/\*------------------------

Typical usage:

@Library('MicroserviceBuilder') \_microserviceBuilderPipeline { image = 'microservice-test'}

The following parameters may also be specified in your Jenkinsfile:

The convention used below is 'parameter' = 'default value' - description

Image = no default value - image name must be specified in your Jenkinsfile

build = 'true' - any value other than 'true' == false

deploy = 'true' - any value other than 'true' == false

gitOptions = '' - any Git config options to use, for example you may wish to provide

--global http.sslVerify false to permit self-signed certificates if you're hosting your own SCM

Maven projects only:

mvnCommands = 'package' - builds project by default, other Maven commands can be specified

test = 'true' - 'mvn verify' is run if this value is 'true' and a pom.xml exists

debug = 'false' - resources created during tests are deleted unless this value is set to 'true'

chartFolder = 'chart' - chart folder to be used for testing only

libertyLicenseJarName = '' - Liberty license jar name to use

These are the names of images to be downloaded from https://hub.docker.com/.

mavenImage = 'maven:3.6.0-jdk-8-alpine'

dockerImage = 'docker:18.06.1-ce'

kubectlImage = 'ibmcom/microclimate-utils:1901'

helmImage = 'ibmcom/microclimate-k8s-helm:v2.12.3'

-------------------------\*/

import com.cloudbees.groovy.cps.NonCPS

import java.io.File

import java.util.UUID

import groovy.json.JsonOutput;

import groovy.json.JsonSlurperClassic;

def call(body) {

def config = [:]

// Parameter expansion works after the call to body() below.

// See https://jenkins.io/doc/book/pipeline/shared-libraries/ 'Defining a more structured DSL'

body.resolveStrategy = Closure.DELEGATE\_FIRST

body.delegate = config

body()

print "microserviceBuilderPipeline : config = ${config}"

// User configurable options

def image = config.image

def build = (config.build ?: env.BUILD ?: "true").toBoolean()

def deploy = (config.deploy ?: env.DEPLOY ?: "true").toBoolean()

def mvnCommands = (config.mvnCommands == null) ? 'package' : config.mvnCommands

def test = (config.test ?: (env.TEST ?: "false").trim()).toLowerCase() == 'true'

def debug = (config.debug ?: (env.DEBUG ?: "false").trim()).toLowerCase() == 'true'

def userSpecifiedChartFolder = config.chartFolder

def chartFolder = userSpecifiedChartFolder ?: ((env.CHART\_FOLDER ?: "").trim() ?: 'chart')

def libertyLicenseJarName = config.libertyLicenseJarName ?: (env.LIBERTY\_LICENSE\_JAR\_NAME ?: "").trim()

def extraGitOptions = config.gitOptions ?: (env.EXTRA\_GIT\_OPTIONS ?: "").trim()

def namespace = (config.namespace ?: env.NAMESPACE ?: "").trim()

def maven = (config.mavenImage == null) ? 'maven:3.6.0-jdk-8-alpine' : config.mavenImage

def docker = (config.dockerImage == null) ? 'docker:18.06.1-ce' : config.dockerImage

def kubectl = (config.kubectlImage == null) ? 'ibmcom/microclimate-utils:1901' : config.kubectlImage

def helm = (config.helmImage == null) ? 'ibmcom/microclimate-k8s-helm:v2.12.3' : config.helmImage

// Internal

def registry = (env.REGISTRY ?: "").trim()

if (registry && !registry.endsWith('/')) registry = "${registry}/"

def registrySecret = (env.REGISTRY\_SECRET ?: "").trim()

def serviceAccountName = (env.SERVICE\_ACCOUNT\_NAME ?: "default").trim()

def mcReleaseName = (env.RELEASE\_NAME).toUpperCase()

def helmSecret = (env.HELM\_SECRET ?: "").trim()

def libertyLicenseJarBaseUrl = (env.LIBERTY\_LICENSE\_JAR\_BASE\_URL ?: "").trim()

def mavenSettingsConfigMap = env.MAVEN\_SETTINGS\_CONFIG\_MAP?.trim()

def alwaysPullImage = (env.ALWAYS\_PULL\_IMAGE == null) ? true : env.ALWAYS\_PULL\_IMAGE.toBoolean()

def helmTlsOptions = " --tls --tls-ca-cert=/msb\_helm\_sec/ca.pem --tls-cert=/msb\_helm\_sec/cert.pem --tls-key=/msb\_helm\_sec/key.pem "

print "microserviceBuilderPipeline: image=${image} build=${build} deploy=${deploy} mvnCommands=${mvnCommands} \

test=${test} debug=${debug} chartFolder=${chartFolder} libertyLicenseJarName=${libertyLicenseJarName} \

registry=${registry} registrySecret=${registrySecret} serviceAccountName=${serviceAccountName} \

mcReleaseName=${mcReleaseName} namespace=${namespace} helmSecret=${helmSecret} libertyLicenseJarBaseUrl=${libertyLicenseJarBaseUrl} \

mavenSettingsConfigMap=${mavenSettingsConfigMap} alwaysPullImage=${alwaysPullImage} helmTlsOptions=${helmTlsOptions} \

maven=${maven} docker=${docker} kubectl=${kubectl} helm=${helm}"

printTime("In the pipeline")

def jobName = (env.JOB\_BASE\_NAME)

// E.g. JOB\_NAME=default/myproject/master

def jobNameSplit = env.JOB\_NAME.split("/")

def projectNamespace = jobNameSplit[0]

def projectName = jobNameSplit[1]

def branchName = jobNameSplit[2]

def testDeployAttempt = 1 // declare here as we'll use again later: don't run tests if it didn't deploy

def verifyAttempt = 1 // we'll need this variable later too: fail the build if we didn't deploy the test release, or tests failed

// We won't be able to get hold of registrySecret if Jenkins is running in a non-default namespace that is not the deployment namespace.

// In that case we'll need the registrySecret to have been ported over, perhaps during pipeline install.

// Only mount registry secret if it's present

def volumes = [ hostPathVolume(hostPath: '/var/run/docker.sock', mountPath: '/var/run/docker.sock') ]

if (registrySecret) {

volumes += secretVolume(secretName: registrySecret, mountPath: '/msb\_reg\_sec')

}

if (mavenSettingsConfigMap) {

volumes += configMapVolume(configMapName: mavenSettingsConfigMap, mountPath: '/msb\_mvn\_cfg')

}

if (helmSecret) {

volumes += secretVolume(secretName: helmSecret, mountPath: '/msb\_helm\_sec')

}

print "microserviceBuilderPipeline: volumes = ${volumes}"

print "microserviceBuilderPipeline: helmSecret: ${helmSecret}"

podTemplate(

label: 'microclimatePod',

inheritFrom: 'default',

serviceAccount: serviceAccountName,

containers: [

containerTemplate(name: 'maven', image: maven, ttyEnabled: true, command: 'cat'),

containerTemplate(name: 'docker', image: docker, command: 'cat', ttyEnabled: true,

envVars: [

containerEnvVar(key: 'DOCKER\_API\_VERSION', value: '1.23.0')

]),

containerTemplate(name: 'kubectl', image: kubectl, ttyEnabled: true, command: 'cat'),

containerTemplate(name: 'helm', image: helm, ttyEnabled: true, command: 'cat')

],

volumes: volumes

) {

node('microclimatePod') {

printTime("In the node microclimatePod code")

def gitCommit

def previousCommit

def gitCommitMessage

def fullCommitID

print "mcReleaseName=${mcReleaseName} projectNamespace=${projectNamespace} projectName=${projectName} branchName=${branchName}"

devopsHost = sh(script: "echo \$${mcReleaseName}\_IBM\_MICROCLIMATE\_DEVOPS\_SERVICE\_HOST", returnStdout: true).trim()

devopsPort = sh(script: "echo \$${mcReleaseName}\_IBM\_MICROCLIMATE\_DEVOPS\_SERVICE\_PORT", returnStdout: true).trim()

devopsEndpoint = "https://${devopsHost}:${devopsPort}"

def imageTag = null

def helmInitialized = false // Lazily initialize Helm but only once

stage ('Extract') {

try {

printTime("In the extract stage")

if (extraGitOptions) {

echo "Extra Git options found, setting Git config options to include ${extraGitOptions}"

configSet = sh(script: "git config ${extraGitOptions}", returnStdout: true)

}

if (fileExists('/var/run/secrets/kubernetes.io/serviceaccount/ca.crt')) {

echo "Adding local cluster certificate"

sh(script: "git config --global http.sslCAInfo /var/run/secrets/kubernetes.io/serviceaccount/ca.crt", returnStdout: true)

}

checkout scm

printTime("checkout scm done")

fullCommitID = sh(script: 'git rev-parse HEAD', returnStdout: true).trim()

gitCommit = sh(script: 'git rev-parse --short HEAD', returnStdout: true).trim()

previousCommitStatus = sh(script: 'git rev-parse -q --short HEAD~1', returnStatus: true)

// If no previous commit is found, below commands need not run but build should continue

// Only run when a previous commit exists to avoid pipeline fail on exit code

if (previousCommitStatus == 0){

previousCommit = sh(script: 'git rev-parse -q --short HEAD~1', returnStdout: true).trim()

echo "Previous commit exists: ${previousCommit}"

}

gitCommitMessage = sh(script: 'git log --format=%B -n 1 ${gitCommit}', returnStdout: true)

gitCommitMessage = gitCommitMessage.replace("'", "\'");

echo "Git commit message is: ${gitCommitMessage}"

echo "Checked out git commit ${gitCommit}"

} catch(Exception ex) {

print "Error in Extract: " + ex.toString()

notifyDevops(gitCommit, fullCommitID, registry + image, imageTag,

branchName, "build", projectName, projectNamespace, env.BUILD\_NUMBER.toInteger(), "FAILED")

error "Stop execution: " + ex.toString()

}

}

if (build) {

if (fileExists('pom.xml')) {

stage ('Maven Build') {

try {

container ('maven') {

printTime("Starting maven build")

def mvnCommand = "mvn -B"

if (mavenSettingsConfigMap) {

mvnCommand += " --settings /msb\_mvn\_cfg/settings.xml"

}

mvnCommand += " ${mvnCommands}"

sh mvnCommand

printTime("Done Maven build")

}

} catch(Exception ex) {

print "Error in Maven build:" + ex.toString()

notifyDevops(gitCommit, fullCommitID, registry + image, imageTag,

branchName, "build", projectName, projectNamespace, env.BUILD\_NUMBER.toInteger(), "FAILED")

error "Stop execution: " + ex.toString()

}

}

}

if (fileExists('Dockerfile')) {

if (fileExists('Package.swift')) {

echo "Detected Swift project with a Dockerfile..."

echo "Checking for runtime image..."

// Remember that grep returns 0 if it's there, 1 if not

def containsRuntimeImage = sh(returnStatus: true, script: "grep 'ibmcom/swift-ubuntu-runtime' Dockerfile")

echo "containsRuntimeImage: ${containsRuntimeImage}"

echo "Checking for a build command..."

def containsBuildCommand = sh(returnStatus: true, script: "grep 'swift build' Dockerfile")

echo "containsBuildCommand: ${containsBuildCommand}"

echo "Checking for microclimate.override=false..."

// Don't do anything with the Dockerfile if we detect this string

def hasOverride = sh(returnStatus: true, script: "grep 'microclimate.override=false' Dockerfile")

echo "hasOverride: ${hasOverride}"

// 0 = true, 1 = false! Would be good to use .toBoolean and make this easier to read

// Modify if there's a runtime image in the FROM, there's no swift build command, there's no override=false

if (containsRuntimeImage == 0 && containsBuildCommand == 1 && hasOverride == 1) {

echo "Modifying the Dockerfile as the Microclimate pipeline has detected the swift-ubuntu-runtime image and no presence of a swift build command in the Dockerfile! Disable this behaviour with microclimate.override=false anywhere in your Dockerfile"

// Use the dev image so we can build

sh "sed -i 's|FROM ibmcom/swift-ubuntu-runtime|FROM ibmcom/swift-ubuntu|g' Dockerfile"

// Add the build command after "COPY . /swift-project"

sh "sed -i '\\/COPY . \\/swift-project/a RUN cd \\/swift-project && swift build -c release' Dockerfile"

// Just run the project they've built: replace their cmd with a simpler one

sh "sed -i 's|cd /swift-project \\&\\& .build-ubuntu/release.\*|cd /swift-project \\&\\& swift run\" ]|g' Dockerfile"

def fileContents = sh(returnStdout: true, script: "cat Dockerfile")

print "Modified Dockerfile is as follows..."

print "${fileContents}"

}

}

stage ('Docker Build') {

try {

container ('docker') {

printTime("About to Docker build")

imageTag = gitCommit

def buildCommand = "docker build -t ${image}:${imageTag} "

buildCommand += "--label org.label-schema.schema-version=\"1.0\" "

def scmUrl = scm.getUserRemoteConfigs()[0].getUrl()

buildCommand += "--label org.label-schema.vcs-url=\"${scmUrl}\" "

buildCommand += "--label org.label-schema.vcs-ref=\"${gitCommit}\" "

buildCommand += "--label org.label-schema.name=\"${image}\" "

def buildDate = sh(returnStdout: true, script: "date -Iseconds").trim()

buildCommand += "--label org.label-schema.build-date=\"${buildDate}\" "

if (alwaysPullImage) {

buildCommand += " --pull=true"

}

if (previousCommit) {

buildCommand += " --cache-from ${registry}${image}:${previousCommit}"

}

if (libertyLicenseJarBaseUrl) {

if (readFile('Dockerfile').contains('LICENSE\_JAR\_URL')) {

buildCommand += " --build-arg LICENSE\_JAR\_URL=" + libertyLicenseJarBaseUrl

if (!libertyLicenseJarBaseUrl.endsWith("/")) {

buildCommand += "/"

}

buildCommand += libertyLicenseJarName

}

}

buildCommand += " ."

if (registrySecret) {

sh "ln -s -f /msb\_reg\_sec/.dockercfg /home/jenkins/.dockercfg"

sh "mkdir -p /home/jenkins/.docker"

sh "ln -s -f /msb\_reg\_sec/.dockerconfigjson /home/jenkins/.docker/config.json"

}

printTime("About to do build command")

sh buildCommand

printTime("Done the build command")

if (registry) {

sh "docker tag ${image}:${imageTag} ${registry}${image}:${imageTag}"

printTime("Pushing to Docker registry")

sh "docker push ${registry}${image}:${imageTag}"

printTime("Done pushing to Docker registry")

}

}

} catch(Exception ex) {

print "Error in Docker build: " + ex.toString()

notifyDevops(gitCommit, fullCommitID, registry + image, imageTag,

branchName, "build", projectName, projectNamespace, env.BUILD\_NUMBER.toInteger(), "FAILED")

error "Stop execution: " + ex.toString()

}

}

}

}

def realChartFolder = null

def testsAttempted = false

if (fileExists(chartFolder)) {

// find the likely chartFolder location

realChartFolder = getChartFolder(userSpecifiedChartFolder, chartFolder)

def yamlContent = "image:"

yamlContent += "\n repository: ${registry}${image}"

if (imageTag) yamlContent += "\n tag: \\\"${imageTag}\\\""

sh "echo \"${yamlContent}\" > pipeline.yaml"

}

if (test && fileExists('pom.xml') && realChartFolder != null && fileExists(realChartFolder)) {

stage ('Verify') {

printTime("In verify stage")

testsAttempted = true

testNamespace = "testns-${env.BUILD\_ID}-" + UUID.randomUUID()

echo "testing against namespace " + testNamespace

String tempHelmRelease = (image + "-" + testNamespace)

// Name cannot end in '-' or be longer than 53 chars

while (tempHelmRelease.endsWith('-') || tempHelmRelease.length() > 53) tempHelmRelease = tempHelmRelease.substring(0,tempHelmRelease.length()-1)

container ('kubectl') {

printTime("In kubectl container")

def testNSCreationAttempt = sh(returnStatus: true, script: "kubectl create namespace ${testNamespace} > ns\_creation\_attempt.txt")

if (testNSCreationAttempt != 0) {

echo "Warning, did not create the test namespace successfully, error code is: ${testNSCreationAttempt}"

}

printFromFile("ns\_creation\_attempt.txt")

def testNSLabelAttempt = sh(returnStatus: true, script: "kubectl label namespace ${testNamespace} test=true > label\_attempt.txt")

if (testNSLabelAttempt != 0) {

echo "Warning, did not label the test namespace ${testNamespace} successfully, error code is: ${testNSLabelAttempt}"

}

printFromFile("label\_attempt.txt")

if (registrySecret) {

giveRegistryAccessToNamespace (testNamespace, registrySecret)

}

}

if (!helmInitialized) {

printTime("Init helm")

initalizeHelm ()

helmInitialized = true

printTime("Done with init helm")

}

container ('helm') {

echo "Attempting to deploy the test release"

printTime("About to Helm install as part of verify")

def deployCommand = "helm install ${realChartFolder} --wait --set test=true --values pipeline.yaml --namespace ${testNamespace} --name ${tempHelmRelease}"

if (fileExists("chart/overrides.yaml")) {

deployCommand += " --values chart/overrides.yaml"

}

if (helmSecret) {

echo "Adding --tls to your deploy command"

deployCommand += helmTlsOptions

}

printTime("About to deploy test release")

testDeployAttempt = sh(script: "${deployCommand} > deploy\_attempt.txt", returnStatus: true)

printTime("Done deploying test release")

if (testDeployAttempt != 0) {

echo "Warning, did not deploy the test release into the test namespace successfully, error code is: ${testDeployAttempt}"

echo "This build will be marked as a failure: halting after the deletion of the test namespace."

}

printFromFile("deploy\_attempt.txt")

}

container ('maven') {

try {

// We have a test release that we can run our Maven tests on

printTime("In Maven container to run tests with")

if (testDeployAttempt == 0) {

def mvnCommand = "mvn -B -Dnamespace.use.existing=${testNamespace} -Denv.init.enabled=false"

if (mavenSettingsConfigMap) {

mvnCommand += " --settings /msb\_mvn\_cfg/settings.xml"

}

mvnCommand += " verify"

printTime("About to verify")

verifyAttempt = sh(script: "${mvnCommand} > verify\_attempt.txt", returnStatus: true)

if (verifyAttempt != 0) {

echo "Warning, did not run ${mvnCommand} successfully, error code is: ${verifyAttempt}"

}

printTime("Done the verify")

printFromFile("verify\_attempt.txt")

} else {

echo "Not running tests as we detected that your test release failed to deploy"

}

} finally {

step([$class: 'JUnitResultArchiver', allowEmptyResults: true, testResults: '\*\*/target/failsafe-reports/\*.xml'])

step([$class: 'ArtifactArchiver', artifacts: '\*\*/target/failsafe-reports/\*.txt', allowEmptyArchive: true])

if (!debug) {

container ('kubectl') {

if (fileExists(realChartFolder)) {

container ('helm') {

printTime("About to helm delete")

def deleteCommand = "helm delete ${tempHelmRelease} --purge"

if (helmSecret) {

echo "adding --tls"

deleteCommand += helmTlsOptions

}

// Until this is done, we can't get both stdout and the status code... https://issues.jenkins-ci.org/browse/JENKINS-44930?page=com.atlassian.jira.plugin.system.issuetabpanels%3Acomment-tabpanel&showAll=true

printTime("About to delete test release")

def deletionAttempt = sh(script: "$deleteCommand > delete\_test\_release\_attempt.txt", returnStatus: true)

if (deletionAttempt != 0) {

echo "Did not delete the test Helm release, error code from ${deleteCommand} is: ${deletionAttempt}"

}

printTime("Done attempting to delete test release")

printFromFile("delete\_test\_release\_attempt.txt")

}

}

// Intentionally do this as the final step in here so we can actually delete it

// A namespace will not be removed if there's a Kube resource still active in there

printTime("Attempting to delete test namespace")

def testNSDeletionAttempt = sh(script: "kubectl delete namespace ${testNamespace} > delete\_test\_namespace\_attempt.txt", returnStatus: true)

if (testNSDeletionAttempt != 0) {

echo "Did not delete the test namespace ${testNamespace} successfully, error code is: ${testNSDeletionAttempt}"

}

printTime("Done attempting to delete test namespace")

printFromFile("delete\_test\_namespace\_attempt.txt")

}

}

}

}

}

}

def result="commitID=${gitCommit}\\n" +

"fullCommit=${fullCommitID}\\n" +

"commitMessage=${gitCommitMessage}\\n" +

"registry=${registry}\\n" +

"image=${image}\\n" +

"imageTag=${imageTag}"

sh "echo \"${result}\" > buildData.txt"

printTime("About to archive artifacts")

archiveArtifacts 'buildData.txt'

printTime("Done archiving artifacts")

// tests are enabled and yet something went wrong (e.g. didn't deploy the test release, or tests failed)? Fail the build

echo "Test is " + test + ", tests attempted: " + testsAttempted

// Pipelines are created with test = true as a default from the Microclimate Helm chart.

// If tests were attempted, and then a problem happened (tests failed, or it didn't deploy, fail the build.

// testsAttempted is set when we enter our testing block: which currently only supports Maven projects.

if (testsAttempted) {

echo "Result of verification is " + verifyAttempt

echo "Result of the test deploy attempt is: " + testDeployAttempt

echo "If either of these values are not 0, we will fail the build"

if (verifyAttempt != 0 || testDeployAttempt != 0) {

def message = "Marking the build as a failed one: test was set to true " +

"and a non-zero return code was returned when running the verify stage in this pipeline. " +

"This indicates there are test failures to investigate or the test release did not deploy. No further pipeline code will be run."

error(message) // this fails the build with an error

}

}

echo "Deploy is " + deploy

if (deploy) {

if (!helmInitialized) {

initalizeHelm ()

helmInitialized = true

}

printTime("About to notify devops")

echo "Notifying Devops"

stage ('Notify Devops') {

notifyDevops(gitCommit, fullCommitID, registry + image, imageTag,

branchName, "build", projectName, projectNamespace, env.BUILD\_NUMBER.toInteger(), "")

}

printTime("Done notifying devops")

}

}

}

}

def printTime(String message) {

time = new Date().format("ddMMyy.HH:mm.ss", TimeZone.getTimeZone('Europe/Amsterdam'))

println "Timing, $message: $time"

}

def printFromFile(String fileName) {

def output = readFile(fileName).trim()

echo output

}

def notifyDevops (String gitCommit, String fullCommitID, String image,

String imageTag, String branchName, String triggerType, String projectName, String projectNamespace, Integer buildNumber, String status) {

notificationEndpoint="${devopsEndpoint}/v1/namespaces/${projectNamespace}/projects/${projectName}/notifications"

print "Poking the notification API at ${notificationEndpoint}, parameters..."

print "gitCommit=${gitCommit}, fullCommitID=${fullCommitID}, image: ${image} \

imageTag=${imageTag}, branchName=${branchName}, triggerType=${triggerType} \

buildNumber=${buildNumber}"

def notificationData = [

chart: [gitCommit: gitCommit, fullCommit: fullCommitID],

overrides: [image: [repository: image, tag: imageTag]],

trigger: [type: triggerType, branch: branchName],

clusterConfigSecret: "",

namespace: "",

status: status,

buildNumber: buildNumber

]

def payload = JsonOutput.toJson(notificationData)

notification = [ 'bash', '-c', "curl -v -k -X POST -H \"Content-Type: application/json\" -d '${payload}' $notificationEndpoint" ].execute().text

print "Devops notification response: ${notification}"

}

def initalizeHelm () {

container ('helm') {

sh "helm init --skip-refresh --client-only"

}

}

/\*

We have a (temporary) namespace that we want to grant ICP registry access to.

String namespace: target namespace

1. Port registrySecret into a temporary namespace

2. Modify 'default' serviceaccount to use ported registrySecret.

\*/

def giveRegistryAccessToNamespace (String namespace, String registrySecret) {

sh "kubectl get secret ${registrySecret} -o json | sed 's/\"namespace\":.\*\$/\"namespace\": \"${namespace}\",/g' | kubectl create -f -"

sh "kubectl patch serviceaccount default -p '{\"imagePullSecrets\": [{\"name\": \"${registrySecret}\"}]}' --namespace ${namespace}"

}

def getChartFolder(String userSpecified, String currentChartFolder) {

def newChartLocation = ""

if (userSpecified) {

print "User defined chart location specified: ${userSpecified}"

return userSpecified

} else {

print "Finding actual chart folder below ${env.WORKSPACE}/${currentChartFolder}..."

def fp = new hudson.FilePath(Jenkins.getInstance().getComputer(env['NODE\_NAME']).getChannel(), env.WORKSPACE + "/" + currentChartFolder)

def dirList = fp.listDirectories()

if (dirList.size() > 1) {

print "More than one directory in ${env.WORKSPACE}/${currentChartFolder}..."

print "Directories found are:"

def yamlList = []

for (d in dirList) {

print "${d}"

def fileToTest = new hudson.FilePath(d, "Chart.yaml")

if (fileToTest.exists()) {

yamlList.add(d)

}

}

if (yamlList.size() > 1) {

print "-----------------------------------------------------------"

print "\*\*\* More than one directory with Chart.yaml in ${env.WORKSPACE}/${currentChartFolder}."

print "\*\*\* Please specify chart folder to use in your Jenkinsfile."

print "\*\*\* Returning null."

print "-----------------------------------------------------------"

return null

} else {

if (yamlList.size() == 1) {

newChartLocation = currentChartFolder + "/" + yamlList.get(0).getName()

print "Chart.yaml found in ${newChartLocation}, setting as realChartFolder"

return newChartLocation

} else {

print "-----------------------------------------------------------"

print "\*\*\* No sub directory in ${env.WORKSPACE}/${currentChartFolder} contains a Chart.yaml, returning null"

print "-----------------------------------------------------------"

return null

}

}

} else {

if (dirList.size() == 1) {

def chartFile = new hudson.FilePath(dirList.get(0), "Chart.yaml")

newChartLocation = currentChartFolder + "/" + dirList.get(0).getName()

if (chartFile.exists()) {

print "Only one child directory found, setting realChartFolder to: ${newChartLocation}"

return newChartLocation

} else {

print "-----------------------------------------------------------"

print "\*\*\* Chart.yaml file does not exist in ${newChartLocation}, returning null"

print "-----------------------------------------------------------"

return null

}

} else {

print "-----------------------------------------------------------"

print "\*\*\* Chart directory ${env.WORKSPACE}/${currentChartFolder} has no subdirectories, returning null"

print "-----------------------------------------------------------"

return null

}

}

}

}

Jenkinsfile (Scripted Pipeline for Adire E-commers)

Node {

Stage (‘Code Get from GitHub’) {

//

Stage (‘Build’) {

//

}

Stage (‘test’) {

//

}

Stage (‘deploy to Tomcat’) {

//

}

}

Commit to tomcat

import grails.util.BuildScope

scriptScope = BuildScope.WAR

includeTargets << grailsScript("\_GrailsWar")

ant.taskdef(name: "deploy", classname: "org.apache.catalina.ant.DeployTask")

ant.taskdef(name: "list", classname: "org.apache.catalina.ant.ListTask")

ant.taskdef(name: "undeploy", classname: "org.apache.catalina.ant.UndeployTask")

target(tomcat: '''\

Script used to interact with remote Tomcat. The following subcommands are available:

grails tomcat deploy - Deploy to a tomcat server

grails tomcat undeploy - Undeploy from a tomcat server

''') {

depends(parseArguments, compile, createConfig)

String cmd = argsMap.params ? argsMap.params[0] : 'deploy'

argsMap.params.clear()

String user = config.tomcat.deploy.username ?: 'manager'

String pass = config.tomcat.deploy.password ?: 'secret'

String url = config.tomcat.deploy.url ?: 'http://localhost:8080/manager'

switch (cmd) {

case 'deploy':

war()

println "Deploying application $serverContextPath to Tomcat"

deploy(war: warName, url: url, path: serverContextPath, username: user, password: pass)

break

case 'list':

list(url: url, username: user, password: pass)

break

case 'undeploy':

configureServerContextPath()

println "Undeploying application $serverContextPath from Tomcat"

println '''\

NOTE: If you experience a classloading error during undeployment you need to take the following steps:

\* Pass this system argument to Tomcat: -Dorg.apache.catalina.loader.WebappClassLoader.ENABLE\_CLEAR\_REFERENCES=false

See http://tomcat.apache.org/tomcat-8.0-doc/config/systemprops.html for more information

'''

undeploy(url: url, path: serverContextPath, username: user, password: pass)

}

}

setDefaultTarget 'tomcat'

SCRIPT 2

import grails.util.BuildScope

scriptScope = BuildScope.WAR

includeTargets << grailsScript("\_GrailsWar")

ant.taskdef(name: "deploy", classname: "org.apache.catalina.ant.DeployTask")

ant.taskdef(name: "list", classname: "org.apache.catalina.ant.ListTask")

ant.taskdef(name: "undeploy", classname: "org.apache.catalina.ant.UndeployTask")

target(tomcat: '''\

Script used to interact with remote Tomcat. The following subcommands are available:

grails tomcat deploy - Deploy to a tomcat server

grails tomcat undeploy - Undeploy from a tomcat server

''') {

depends(parseArguments, compile, createConfig)

String cmd = argsMap.params ? argsMap.params[0] : 'deploy'

argsMap.params.clear()

String user = config.tomcat.deploy.username ?: 'manager'

String pass = config.tomcat.deploy.password ?: 'secret'

String url = config.tomcat.deploy.url ?: 'http://localhost:8080/manager'

switch (cmd) {

case 'deploy':

war()

println "Deploying application $serverContextPath to Tomcat"

deploy(war: warName, url: url, path: serverContextPath, username: user, password: pass)

break

case 'list':

list(url: url, username: user, password: pass)

break

case 'undeploy':

configureServerContextPath()

println "Undeploying application $serverContextPath from Tomcat"

println '''\

NOTE: If you experience a classloading error during undeployment you need to take the following steps:

\* Pass this system argument to Tomcat: -Dorg.apache.catalina.loader.WebappClassLoader.ENABLE\_CLEAR\_REFERENCES=false

See http://tomcat.apache.org/tomcat-8.0-doc/config/systemprops.html for more information

'''

undeploy(url: url, path: serverContextPath, username: user, password: pass)

}

}

setDefaultTarget 'tomcat'