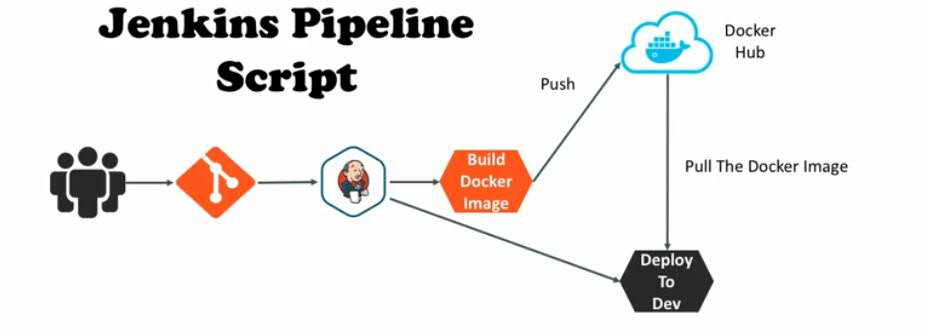
**CI & CD of Docker Application using Jenkins Pipeline Script and pushing image to Docker Hub**

In this document, we are going to learn, how to integrate Docker with Jenkins, how to push and pull image from Docker Hub and run the image using pipeline script. Below screen shot can give better explanation:



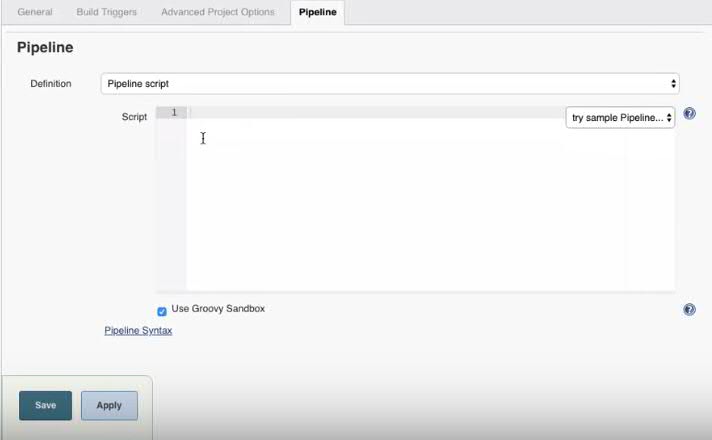
The flow is, first developers will commit their code to GitHub repository. Then we will create a pipeline script in Jenkins, which pulls the latest code from, then compiles the code and creates and Docker Image. If the Docker image is created successfully, then we will do authentication to Docker Hub and push the image to Docker Hub using the same pipeline script. Finally, we will pull and run the Docker Image, what we pulled to Docker Hub. This is the process what we are going to learn in this document and all the flow is done using Jenkins and pipeline script.

To achieve this, we need to create 2 instances in AWS or else 2 virtual Linux machines are also fine.  
  
In the first Linux machine, we have to use Jenkins, where we are going to create the build job using pipeline script.  
The second Linux machine is required to run the Docker image on container, so install docker engine on it and keep ready.

To do above steps, please refer my other Docker documents, to know how to install Docker and Jenkins.

**Creating Pipeline Job in Jenkins:**  
First, install Jenkins on the first Linux machine and login to it. Also install Docker on the first machine, as the Jenkins has to run Docker related commands.

Then create a new job, by clicking on **New Item**. Then enter the job name (Ex**:** **docker-jenkins**) and select the **pipeline** style of project and click on **OK**. Then go to **Pipeline** section, so that we will see a **Script** field like below to write the script.



The pipeline script what we want to write can be written here or else we can create a fine called Jenkins file inside project and move it to Git Hub repository. We can use that file as well to run the pipeline script.

If you observer the above picture, there is an option called **Pipeline Syntax**, this is the option given by Jenkins to generate the pipeline script. We no need to write the complete script by our own, we can take help of Jenkins **Pipeline Syntax** option to write the script.  
  
**Let’s write the pipeline script:**

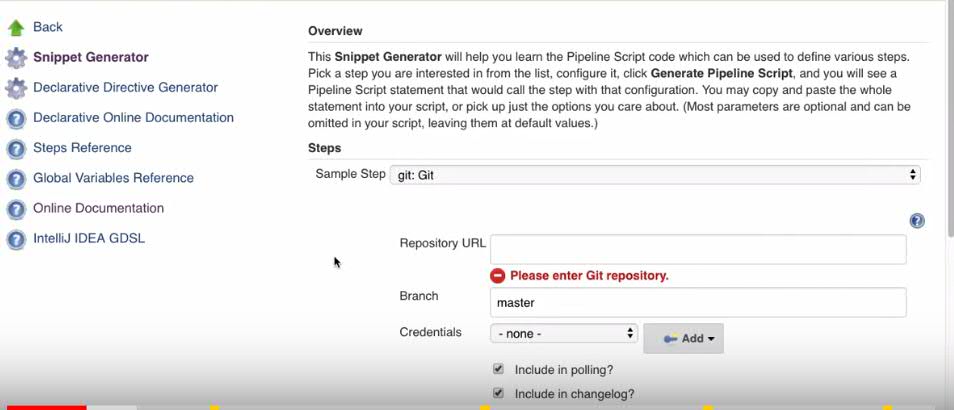
For this we need to have some basic idea about Groovy scripting as we are going to write the script using Groovy.

**Step 1: To check out the code from Git Hub repository**

First, we need to specify the node details on which our script has to run. As I have to run the script on any node and no node restrictions, I am just specifying the **node{ },** otherwise we have to write the node name (**node(“node label”)**) as well, on which node we want to restrict the job has to run.

Then we have to write different stages for each operation, like one stage for git checkout, one stage for maven build, one stage for logging and pushing docker image to Docker Hub. Let’s create our first stage using Pipeline Syntax generator to check out the code from GIT. For this, click on the **Pipeline Syntax** option.

Then you will get options like below.

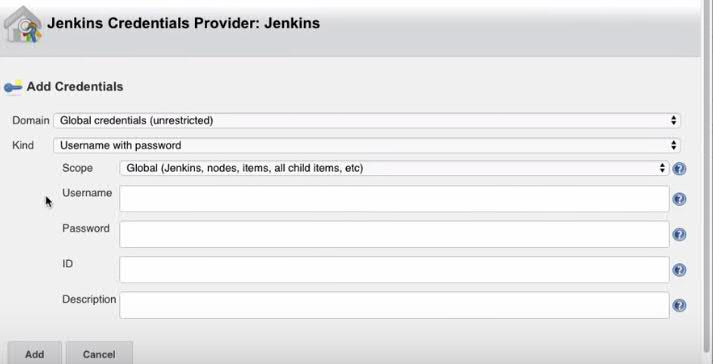


Every operation we have to do, will have a step in Jenkins Pipeline script generator. As we have to check out the code from GIT, then select the Step as **git: GIT**.

Then give the Git Repository URL:

Then give the branch name:

If you already have GIT login credentials, then select those at the Credentials option. If not, then click on Add option.



Then give the credentials, add those and then select then.

Then click on the **Generate Pipeline Script** option, so that Jenkins will create a script for you. Copy the generated script and paste it in the pipeline script stage we are writing.

node{

stage('SCM Checkout'){

git credentialsId: 'git-credentials', url: 'https://github.com/kctechnologies/docker-pipeline

}  
}

Above stage says that, it has to fetch the code from the give git hub repo using the specified credentials. By default it will fetch from master branch.  
To fetch the code from specific branch, then we have to add one more param with its value like below.

node{

stage('SCM Checkout'){

git credentialsId: 'git-credentials', url: 'https://github.com/kctechnologies/docker-pipeline, branch: ‘development’

}  
}

As we mentioned the development as the branch, this script will fetch the code from development branch.

For testing the script, we written so far, we can save the job and try out Build Now option. Then in the console, we can see the code cloning from GIT hub, if we have configured everything correct.

**Step 2: Running maven goal to generate war**

In step 2, we have to build the project using maven to generate a war file. This jar file we are going to use to create a Docker image and push it.   
To do this, we need to create one more stage.

node{

stage('SCM Checkout'){

git credentialsId: 'git-credentials', url: 'https://github.com/kctechnologies/docker-pipeline, branch: 'development'

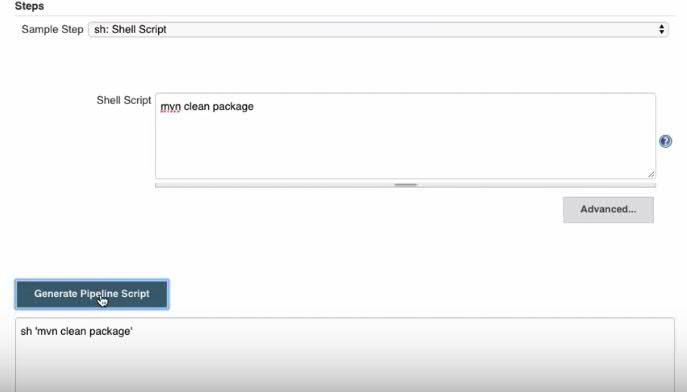
}

stage('Maven Packaging'){

}  
}

Then take the help of Jenkins **Pipeline script** generator to create the script.

In the script generator, we will not have direct step related to Maven, so we will use shell script option (**sh: Shell Script**). Using this shell script, we will run the maven goals. So, give the goal as **mvn clean packag**e and click on **Generate Pipeline Script**.



Copy the generated script and paste in the created stage.

node{

stage('SCM Checkout'){

git credentialsId: 'git-credentials', url: 'https://github.com/kctechnologies/docker-pipeline, branch: ‘development’

}

stage('Maven Packaging'){

sh ‘mvn clean package’

}  
}

Copy this script to Jenkins job and run it for testing.

Your job may fail saying that **mvn command not found** as it is not able to find maven library even though we have installed maven.

To resolve this, we can explicitly mention the path for maven from where our maven command has to run. Let’s see how we can do this.

To do this, first go to **Jenkins** home, then go to **Manage Jenkins**, then go to **Global Tool Configuration**. There go to maven configuration we did.



The Maven configuration name we have given can be used in our pipeline script to point maven home.

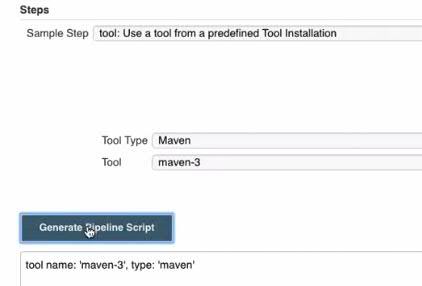
To use it in our script we can take Pipeline Syntax generator.

From the syntax generator, select the **Step: tool: Use a tool from a predefined Tool installation**.

Then it will ask the **Tool Type** and **Tool**.

Tool Type is the type of tool we want, it will list the tools based on the Jenkins plugins we added. In our case as well Maven, then select the Tool Type as **maven**.

Tool indicates the name of the tool. If we have multiple maven configurations with different versions, then it will list all of them. In our case, select the name of the required maven configuration and click on **Generate Pipeline Script**.



So that a script will be generated for us. In that script **tool** indicates the function and **name** and **type** are the its arguments. This **tool** function gives us the **Maven home path**. Copy the generated script and paste in the Stage where we have maven package goal.

stage('Maven Packaging'){

def mvnHome = tool name: 'maven-3', type: 'maven'

def mvnCMD = "${mvnHome}/bin/mvn"

sh "${mvnCMD} clean package"

}

Here, we use **def** which is the Groovy keyword to create a variable and **mvnHome** is the variable name. So that we have the maven home path in the **mvnHome** variable. Also create one more variable **mvnCMD** to get complete path of maven with **bin/mvn**.

Make sure you are using double quotes while reading any variable value. It is recommended in Groovy script to use double quotes while reading the variable values. Then run maven goal from the maven path we got in the **mvnCMD** variable.

Now test your code is working or not, by copying your script to Jenkins job and build it. It will create war file and shows success response if you have done all the configurations correctly.

**Step 3: Building docker image**

For this create one more stage. As there is no direct step for docker, we can use shell script option (sh: Shell Script).

stage('Building docker image'){

sh 'docker build -t kctechnologies/docker-jenkins:1.0.0 .'

}

Here, **docker build -t** is the command to build docker image. **Kctechnologies** is the Docker Hub login user name, **docker-jenkins** is the image name what we want to push to docker hub and **1.0.0** is the image tag (version).

The **.** symbol at the end of the command indicates the location of the Dockerfile. In our case as we have the Dockerfile file at the same location of Jenkinsfile, we using . which indicates current location, else we have to give the path of the Dockerfile.

This is for a public docker hub. For a private Docker registry, we use url with port number in place of username. This we will discuss in detail in my next classes.

**Note:** Make sure your Jenkins user has permissions to run Docker commands.

To test the build script we create, copy the script to Jenkins job and run it. If everything is configured well, then a docker image will be created.

**Step 4: Pushing the created image to Docker Hub**

For this, create one more stage, and add the docker command to push the image to Docker hub.

stage('Pushing Docker image to docker hub'){

sh 'docker push kctechnologies/docker-jenkins:1.0.0'

}

Here, **docker push** is the command to push the image from local to docker hub. Then its all you know, username and image name with tag. This stage will push the docker image we created in previous stage.   
But, to push the image to docker hub, we need to an account in docker hub and before pushing the image, we need to login to docker hub.  
First go and create an account in Docker Hub, if you don’t have.

Now, we will do how to login to docker hub using pipeline script, before pushing the image.

stage('Pushing Docker image to docker hub'){

sh 'docker login -u kctechnologies -p mypassword'

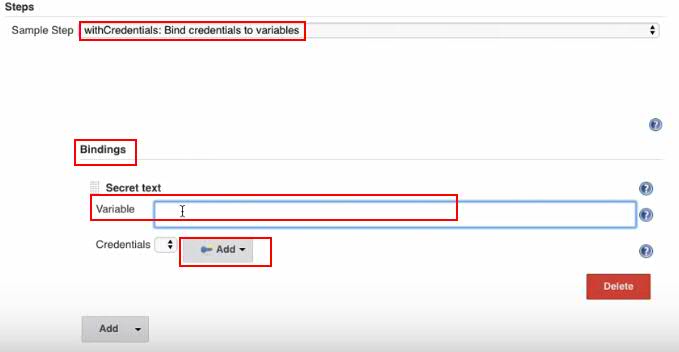
sh 'docker push kctechnologies/docker-jenkins:1.0.0'

}

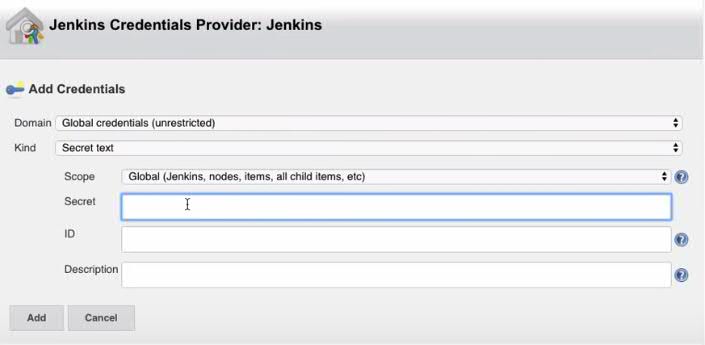
Here we are passing the password directly which is not a recommended way. Actually, we have to make the password secret. For this Jenkins pipeline script will provide an option called binding the credentials.

To get this, first go to Pipeline script generator, select the Step as **withCredentials: Bind credentials to variables.** After selecting, you will get an option called **Bindings**. Using this withCredentails option, we can bind your credentials to a variable and can use the variable in our script, so that we can maintain our password as secret.

So click on the Add option at the **Bindings** option and select the **Secret text** option and give some variable name as we require.



Then click on the Add option, then select the secret text option.



Then give you docker hub password at the **Secret** field. Then give ID and Description for reference purpose.

Then select the created password for the mentioned variable and click on **Generate Pipeline Script** option. So that a script will be generated for us with the withCredentials function.

Copy the script and paste in side our stage.

stage('Pushing docker image to docker hub'){

withCredentials([string(credentialsId: 'docker-passwd', variable: 'dockerHubPasswd')]) {

sh "docker login -u kctechnologies -p ${dockerHubPasswd}"

}

sh 'docker push kctechnologies/docker-jenkins:1.0.0'

}

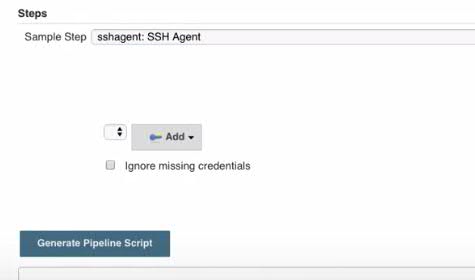
**docker login** is the command to login to docker hub, **-u** is the username and **-p** is the password.

Here, when we call the variable we created (dockerHubPasswd), it will get the secret key we created for docker-passwd variable. In this way we can maintain our passwords secretly. In the login command, we are calling the variable value, so make sure that we are using double quotes “”.

Now, copy this script to our Jenkins Job and run it. If everything is configured well, then it will push the tagged version of image in to docker hub. You can cross check the pushed image in your docker hub registry under **Tags** option.

**Step 5: Running container on our remote server**

For this we have to define one more stage. In that stage we have to use a plugin called sshAgent. Using sshAgent, we can run commands on remote machine. As usual, we will take Jenkins pipeline script generator to use sshAgent. Select the Step as sshagent: SSH Agent.



This sshagent option we will not get by default, we have to add sshagent plugin to Jenkins. Refer my Jenkins document to know how to add plugins to Jenkins.

To give the client credentials, click on the Add option.

As we are not aware of AWS and AWS AMI, for now we will login using pem file. As we are using Aws pem secret file, select the **SSH username with private key** option at the **Kind** option. Then give the **user name** as **ec2-user**.

Then select the **Enter directly** under **private Key** option and paste the content of pem file. Then give some ID and Description for reference purpose. Then add it and, select it and click on **Generate Pipeline Script** option.   
Copy the generated syntax and paste it in the pipeline script stage.

stage('Running docker Container on remote Server'){

sshagent(['remote-server']) {

}

}

So far, we just created a connection to remote server. But that is not enough, and we need to do something more. First let me write it and then explain for better understanding.

stage('Running docker container on remote server'){

def runCommand = 'docker run -p 8080:8080 -d --name docker-jenkins kctechnologies/docker-jenkins:1.0.0'

sshagent(['remote-server']) {

sh "ssh -o StrictHostKeyChecking=no ec2-user@172.22.16.194 ${runCommand}"

}

}

Here we created a variable with the name **runCommand**. **docker run** is the command to create a container out of an image. **-p 8080:8080** indicates the port mapping. **-d** indicates detached mode. **--name docker-jenkins** indicates the name of the container. **kctechnologies/docker-jenkins:1.0.0** indicates the username, image name and its tag.

We created a command to run the image. But we want to run it on remote server. So, call this command using sshagent.

Let me explain what we did inside the sshagent.

**ssh** is the Linux command to do ssh connection. While connecting to remote machine, we may get some popups asking for yes or no, to suppress such things we used **-o StrictHostKeyChecking=no**.

Then **ec2-user@ipaddress** is the username and ipaddress of the remote machine we want to connect and **${runCommand}** is the command we want to run on remote machine. Make sure we have written the commands inside double quotes “”.

Before running the new container, we can remove the existing containers if any. For this we need to create one more stage like below.

stage('Removing old containers'){

try{

def dockerRemove = 'docker rm -f docker-jenkins'

sshagent(['remote-server']) {

sh "ssh -o StrictHostKeyChecking=no ec2-user@172.22.16.194 ${ dockerRemove }"

}

}catch(error){

// write the exception handling code here like sending notification.

}

}

Now, copy the script and paste it in Jenkins job and build the Job.

If everything is configured well, then we will get a success response for the build and we can access out application from browser.

Try to access the application from browser.  
<http://ipaddress:8080> will show the admin console of tomcat.

<http://ipaddress:8080/docker-jenkins-1.0.0/> will show the application home page.