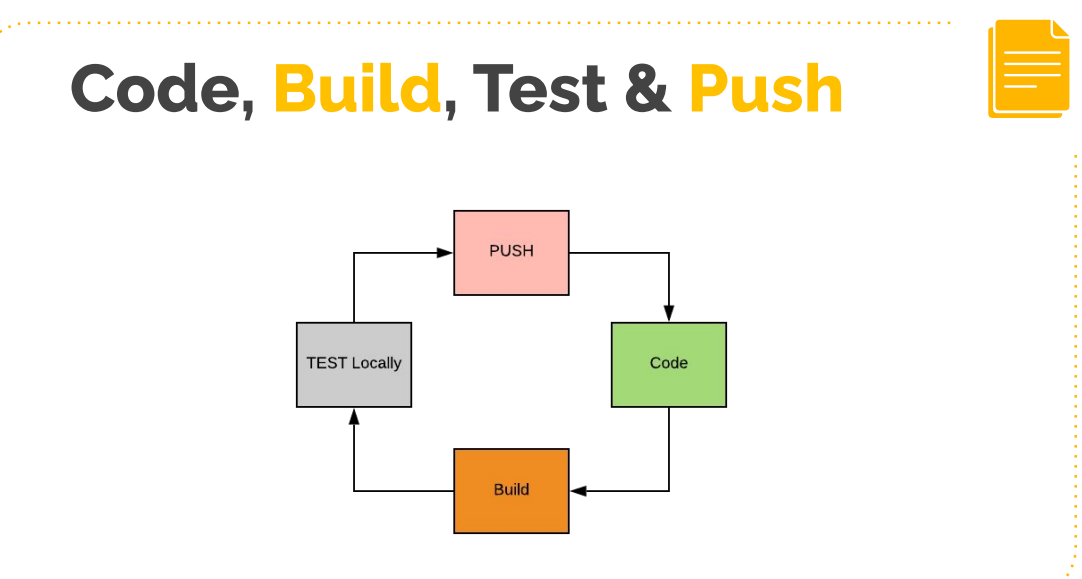
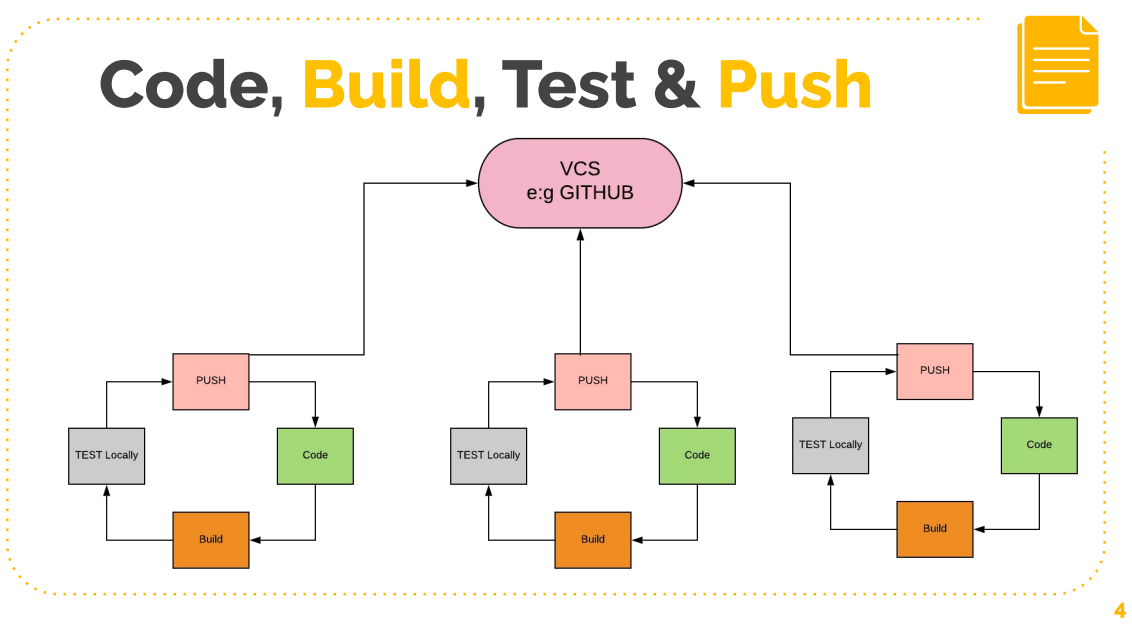
# Jenkins

What is Continuous Integration?

When developer, working on a product development, will code, build the code, test it and push it in the centralized version control system.



Continuous process : right the code, build & test locally, if everything is good then push into the centralized control system like that all the developer will be doing same thing in the team



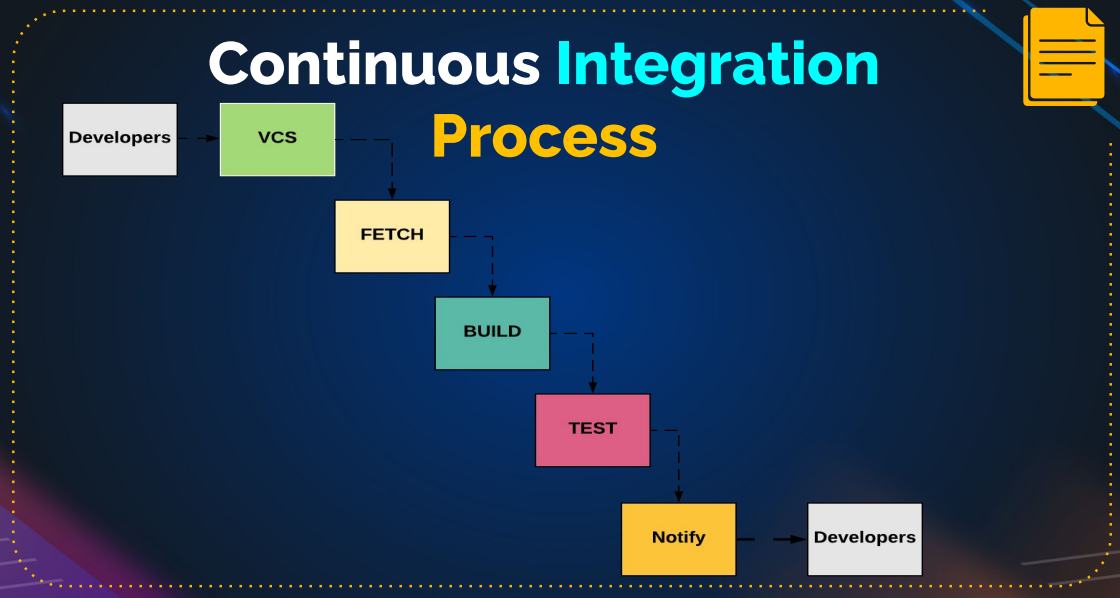
But they is problem in it, well, this is happening many, many times in a day by all the different developers and the code gets marge in the centralized repository system on a regular basis and days past week pass, sometimes even month pass the code keep getting merge and then after some time , if the code is together build and tested it generates a lot of bugs, issues and conflicts.

**That is happening because the code is getting merge, but really not getting integrated**

Developer keeps merging the code to the version control system multiple time in a day and several times in a week/month and there could be so many bugs. well, this code is getting merged and after a long time, when it is tested together, it generated those errors, those bugs, those conflicts and it is very natural , but the problem starts after that everything need to be fixed so developers needs to do a lot of rework, invest a lot of time in fixing all this maybe deadline is coming close/past the deadline

So integration is a painful, but we can resolve it, they is solution for this whenever the code changes in centralized repository or for every single commit that done by the any developer. We build the code, build code for every commit. Build and test the code and this has to done by every commit then this process needs to be automated whenever the code change, the code be fetched, build and tested. If anything is wrong the developer gets notify’s and fix the bug/issues and build the code again. **This process is called Continuous Integration**

the code get’s build & test in automated way is is called continuous integration .

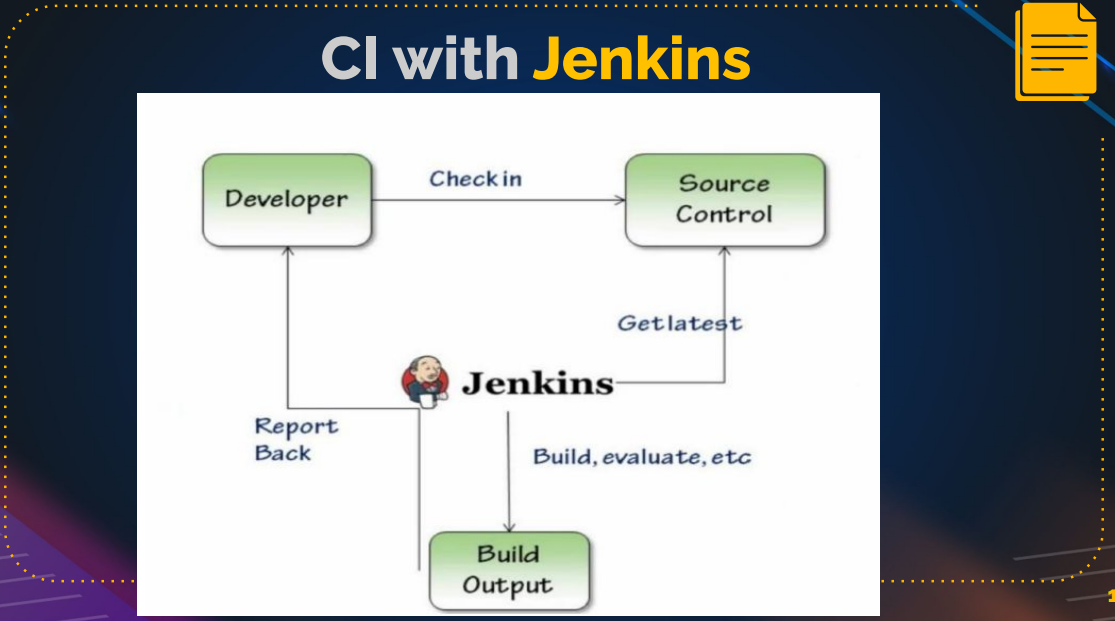


In the process (vcs🡪fectch🡪build🡪test🡪notify )Dev got the error any step the fix the issue and the continuous the process.

So not only just keep merging the code but also for every commit which will test and notify.

If this is a continuous process, then we need to automate this process , if we automate this process this called a continuous integration process and Jenkins & nexus is one of the tools for CI /CD process.

Jenkins:



This as two majar features

1)opensource

2) extensible(Plugins: ● VCS Plugin ● Build Plugin ● Cloud Plugin ● Testing Plugin ● Etc etc etc)

## Jenkins Installation

Prerequisite 1. Java- JRE, JDK 2. Any OS

<https://www.jenkins.io/> - refer the Jenkins links for installation (refer documents)

hands-on: create a ec2 ubuntu instance 🡪 followed the installation steps

ex: steps for ubuntu mahine

#!/bin/bash

sudo apt update

sudo apt install openjdk-17-jdk -y

curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key | sudo tee \

/usr/share/keyrings/jenkins-keyring.asc > /dev/null

echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] \

https://pkg.jenkins.io/debian-stable binary/ | sudo tee \

/etc/apt/sources.list.d/jenkins.list > /dev/null

sudo apt-get update

sudo apt-get install jenkins

## Jobs in Jenkins

They are different of jobs mainly comparing the freestyle vs pipeline as a code

**Jobs :** jobs basically means workload, which Jenkins runs for you

Freestyle jobs : (not recommended now-a-days)

* Graphical jobs
* Learning, understanding & exploring Jenkins.
* Not recommended in real time now

Pipeline As a Code

* Pipleline created in groovy
* Recommended now

# Installing the tools in Jenkins

Now we can create a job in Jenkins, which is going to fetch our source code and build that source by using mvn install command

We required three thing

* Git 🡪 to fetch the source code
* Maven
* Installing the dependency of maven 🡪 jdk

Whenever you are using tools in Jenkins, we need to remember two things

* One is the tool itself(you can install the tools by logging in jenkins instance, and install the tool by using APT commands)
* Plugins

In order to install the tools in **Jenkins 🡪 manage Jenkins 🡪tools🡪add the path for jdk and maven**

Jdk (/usr/lib/jvm/java-17.0-openjdk-amd64)

## Create a new job from freestyle

Add the description

Execute the shell completed like “whoami id pws”

2) echo "i am trying to learn devops from imron" >> jenkins.txt

echo "##################################"

echo "this is my jenkins job $USER"

ls

echo "##################################"

cat jenkins.txt

create a new job for build vprofile using maven

* Job: build 🡪 description build the artifacts from vprofiles
* Add the git url 🡪 https://github.com/hkhcoder/vprofile-project.git -->\*/main
* Jdk 🡪 OracleJDK17(normally jdk11 I had used jdk 17)
* Select :MAVEN3 🡪 clean install (config the version in tools)

If you going to default then you need to **install the maven in server region**

And run the job then it create a artifact

If I am not able to run the job due to low space in ec2 instance them

* Stop the ec2 server 🡪 instance setting 🡪change the instance type(to increase the cpu and ram)

In post build:

We can give the \*\*/\*.war to archive the artifacts

Job : Build\_test

We can copy the configuration one job to other(copy build 🡪 Build\_test)

# **Plugins, Versioning & more**

### **Versioning**

**🡪normally when the build completed then output is generated in the workspace, in the target directory, we are seeing the artifact.**

**Workspace🡪target 🡪 artifact**

🡪every time we run the job; this artifact will get replaced with newer version of artifact. But here we cannot retrieve older version of artifact. so, to preserver all the artifacts, you can do some versioning (we can use some shell commands)

Mkdir -p versions

Cp target/vprofile-v2.war versions/vprofile.V$BUILD\_ID.war

We parameterized so that we can give the version at run time(string parameterized)

Mkdir -p versions

Cp target/vprofile-v2.war versions/vprofile.V$versions.war

Now you will get build with parameters 🡪(it is case sensitive)

But tried avoiding parameter from use

**See the plugins in Jenkins**

**Ex:-** install the plugin for timestamp **🡪 zentimestamp**

Now we can able to see the timestamp in configuration 🡪 yyyymmddhhmm

Mkdir -p versions

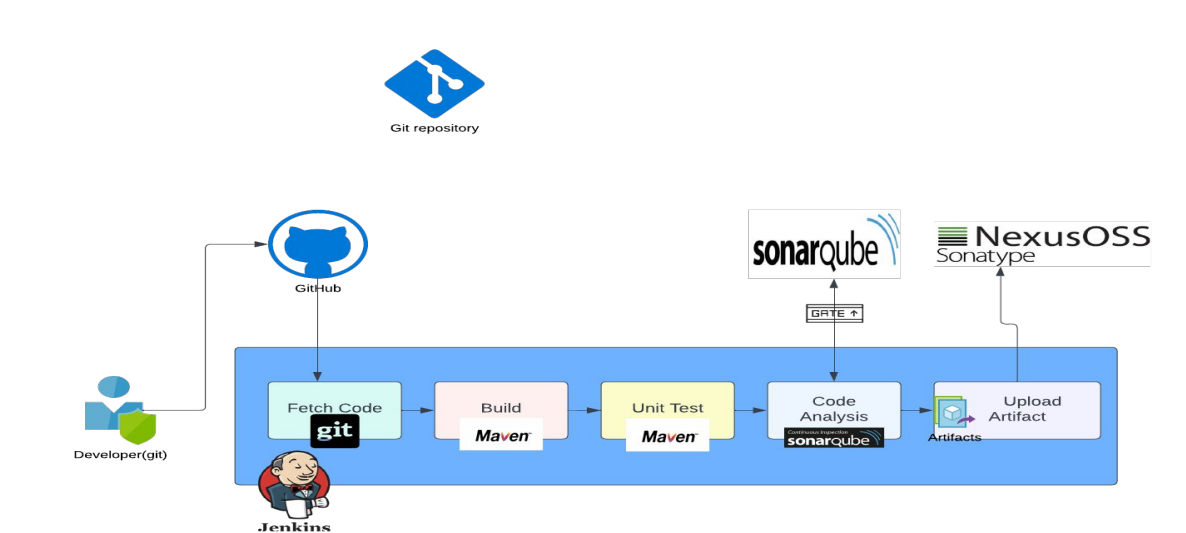
Cp target/vprofile-v2.war versions/vprofile.V$BUILD\_ID -$BUILD\_TIMESTAMP.war

## **Jenkins Set Environment Variables**

When a Jenkins job executes, it sets some environment variables that you may use in your shell script, batch command, Ant script or Maven POM [#1](https://wiki.jenkins.io/display/JENKINS/Building+a+software+project#Buildingasoftwareproject-1). The following table contains a list of all of these environment variables.

|  |  |
| --- | --- |
| **Environment Variable** | **Description** |
| BUILD\_NUMBER | The current build number, such as "153" |
| BUILD\_ID | The current build id, such as "2005-08-22\_23-59-59" (YYYY-MM-DD\_hh-mm-ss, [defunct](https://issues.jenkins-ci.org/browse/JENKINS-26520) since version 1.597) |
| BUILD\_URL | The URL where the results of this build can be found (e.g. http://buildserver/jenkins/job/MyJobName/666/) |
| NODE\_NAME | The name of the node the current build is running on. Equals 'master' for master node. |
| JOB\_NAME | Name of the project of this build. This is the name you gave your job when you first set it up. It's the third column of the Jenkins Dashboard main page. |
| BUILD\_TAG | String of jenkins-${JOB\_NAME}-${BUILD\_NUMBER}. Convenient to put into a resource file, a jar file, etc for easier identification. |
| JENKINS\_URL | Set to the URL of the Jenkins master that's running the build. This value is used by [Jenkins CLI](https://wiki.jenkins.io/display/JENKINS/Jenkins-CLI.html) for example |
| EXECUTOR\_NUMBER | The unique number that identifies the current executor (among executors of the same machine) that's carrying out this build. This is the number you see in the "build executor status", except that the number starts from 0, not 1. |
| JAVA\_HOME | If your job is configured to use a specific JDK, this variable is set to the JAVA\_HOME of the specified JDK. When this variable is set, PATH is also updated to have $JAVA\_HOME/bin. |
| WORKSPACE | The absolute path of the workspace. |
| SVN\_REVISION | For Subversion-based projects, this variable contains the revision number of the module. If you have more than one module specified, this won't be set. |
| CVS\_BRANCH | For CVS-based projects, this variable contains the branch of the module. If CVS is configured to check out the trunk, this environment variable will not be set. |
| GIT\_COMMIT | For Git-based projects, this variable contains the Git hash of the commit checked out for the build (like ce9a3c1404e8c91be604088670e93434c4253f03) (all the GIT\_\* variables require git plugin) |
| GIT\_URL | For Git-based projects, this variable contains the Git url (like git@github.com:user/repo.git or [https://github.com/user/repo.git]) |
| GIT\_BRANCH | For Git-based projects, this variable contains the Git branch that was checked out for the build (normally origin/master) |

Flow of continuous integration pipeline:



Developer(dev’s write the code/make the changes in code and test it locally if they are good with changes, they will push it to a centralized repository like GitHub.)

🡪 so developer will have a tool which integrate with github repository and the code will be committed to github repository as soon as there it a code change Jenkins will detect and fetch the code by using git tool (**Jenkins as git tool and git plugin which will help accomplish this task to fetch the code**)

🡪code pipeline **MAVEN** Using to build the code because we have a java code(our code can be build with maven(vprofile) , we can also use **ANT🡪 ONE BUILD COMPLETE IT WILL GENERATE ARTIFECT**

**🡪** we will conduct the unit test by using maven(unit testing is a part of source code) , need to execute some step that run the testand generate reports mostly in xml format

* Code analysis

Unit test 🡪 it check the unit of the code works or not.

Code analysis 🡪 it check if the code as any vulnerability, are you following the best practices, bug in the code, many other analysis to judge your code.

We will use **sonarqube** scanner , checkstyle to scan the code, this will generate the reports in xml format

* This xml reports will uploaded to sonarqube server , so that we can see the info in graph, charts to see vulnerable, bugs, best particles etc. if the test failed the pipeline will stoped
* If the test completed the we have verified copy of artifacts 🡪 before distribute artifact to be deployed on server, this artifact will properly versioned and store in NEXUS OSS sonartype repository

## Steps for Continuous Integration pipeline

Steps:

* Jenkins setup, Nexus setup & sonarqube setup 🡪 this done by bash script in ec2
* Security group
* Plugins
* Integrate

Nexus

Sonarqube

Write pipeline script

Set notification

# Jenkins, Nexus and SonarQube setup

Nexus

#!/bin/bash

yum install java-1.8.0-openjdk.x86\_64 wget -y

mkdir -p /opt/nexus/

mkdir -p /tmp/nexus/

cd /tmp/nexus/

NEXUSURL="https://download.sonatype.com/nexus/3/latest-unix.tar.gz"

wget $NEXUSURL -O nexus.tar.gz

sleep 10

EXTOUT=`tar xzvf nexus.tar.gz`

NEXUSDIR=`echo $EXTOUT | cut -d '/' -f1`

sleep 5

rm -rf /tmp/nexus/nexus.tar.gz

cp -r /tmp/nexus/\* /opt/nexus/

sleep 5

useradd nexus

chown -R nexus.nexus /opt/nexus

cat <<EOT>> /etc/systemd/system/nexus.service

[Unit]

Description=nexus service

After=network.target

[Service]

Type=forking

LimitNOFILE=65536

ExecStart=/opt/nexus/$NEXUSDIR/bin/nexus start

ExecStop=/opt/nexus/$NEXUSDIR/bin/nexus stop

User=nexus

Restart=on-abort

[Install]

WantedBy=multi-user.target

EOT

echo 'run\_as\_user="nexus"' > /opt/nexus/$NEXUSDIR/bin/nexus.rc

systemctl daemon-reload

systemctl start nexus

systemctl enable nexus

Setup

## Sonar-analysis properties:

sonar.projectKey=vprofile

sonar.projectName=vprofile-repo

sonar.projectVersion=1.0

sonar.sources=src/

sonar.java.binaries=target/test-classes/com/visualpathit/account/controllerTest/

sonar.junit.reportsPath=target/surefire-reports/

sonar.jacoco.reportsPath=target/jacoco.exec

sonar.java.checkstyle.reportPaths=target/checkstyle-result.xml

## sonar setup

#!/bin/bash

cp /etc/sysctl.conf /root/sysctl.conf\_backup

cat <<EOT> /etc/sysctl.conf

vm.max\_map\_count=262144

fs.file-max=65536

ulimit -n 65536

ulimit -u 4096

EOT

cp /etc/security/limits.conf /root/sec\_limit.conf\_backup

cat <<EOT> /etc/security/limits.conf

sonarqube - nofile 65536

sonarqube - nproc 409

EOT

sudo apt-get update -y

sudo apt-get install openjdk-11-jdk -y

sudo update-alternatives --config java

java -version

sudo apt update

wget -q https://www.postgresql.org/media/keys/ACCC4CF8.asc -O - | sudo apt-key add -

sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt/ `lsb\_release -cs`-pgdg main" >> /etc/apt/sources.list.d/pgdg.list'

sudo apt install postgresql postgresql-contrib -y

#sudo -u postgres psql -c "SELECT version();"

sudo systemctl enable postgresql.service

sudo systemctl start postgresql.service

sudo echo "postgres:admin123" | chpasswd

runuser -l postgres -c "createuser sonar"

sudo -i -u postgres psql -c "ALTER USER sonar WITH ENCRYPTED PASSWORD 'admin123';"

sudo -i -u postgres psql -c "CREATE DATABASE sonarqube OWNER sonar;"

sudo -i -u postgres psql -c "GRANT ALL PRIVILEGES ON DATABASE sonarqube to sonar;"

systemctl restart postgresql

#systemctl status -l postgresql

netstat -tulpena | grep postgres

sudo mkdir -p /sonarqube/

cd /sonarqube/

sudo curl -O https://binaries.sonarsource.com/Distribution/sonarqube/sonarqube-8.3.0.34182.zip

sudo apt-get install zip -y

sudo unzip -o sonarqube-8.3.0.34182.zip -d /opt/

sudo mv /opt/sonarqube-8.3.0.34182/ /opt/sonarqube

sudo groupadd sonar

sudo useradd -c "SonarQube - User" -d /opt/sonarqube/ -g sonar sonar

sudo chown sonar:sonar /opt/sonarqube/ -R

cp /opt/sonarqube/conf/sonar.properties /root/sonar.properties\_backup

cat <<EOT> /opt/sonarqube/conf/sonar.properties

sonar.jdbc.username=sonar

sonar.jdbc.password=admin123

sonar.jdbc.url=jdbc:postgresql://localhost/sonarqube

sonar.web.host=0.0.0.0

sonar.web.port=9000

sonar.web.javaAdditionalOpts=-server

sonar.search.javaOpts=-Xmx512m -Xms512m -XX:+HeapDumpOnOutOfMemoryError

sonar.log.level=INFO

sonar.path.logs=logs

EOT

cat <<EOT> /etc/systemd/system/sonarqube.service

[Unit]

Description=SonarQube service

After=syslog.target network.target

[Service]

Type=forking

ExecStart=/opt/sonarqube/bin/linux-x86-64/sonar.sh start

ExecStop=/opt/sonarqube/bin/linux-x86-64/sonar.sh stop

User=sonar

Group=sonar

Restart=always

LimitNOFILE=65536

LimitNPROC=4096

[Install]

WantedBy=multi-user.target

EOT

systemctl daemon-reload

systemctl enable sonarqube.service

#systemctl start sonarqube.service

#systemctl status -l sonarqube.service

apt-get install nginx -y

rm -rf /etc/nginx/sites-enabled/default

rm -rf /etc/nginx/sites-available/default

cat <<EOT> /etc/nginx/sites-available/sonarqube

server{

listen 80;

server\_name sonarqube.groophy.in;

access\_log /var/log/nginx/sonar.access.log;

error\_log /var/log/nginx/sonar.error.log;

proxy\_buffers 16 64k;

proxy\_buffer\_size 128k;

location / {

proxy\_pass http://127.0.0.1:9000;

proxy\_next\_upstream error timeout invalid\_header http\_500 http\_502 http\_503 http\_504;

proxy\_redirect off;

proxy\_set\_header Host \$host;

proxy\_set\_header X-Real-IP \$remote\_addr;

proxy\_set\_header X-Forwarded-For \$proxy\_add\_x\_forwarded\_for;

proxy\_set\_header X-Forwarded-Proto http;

}

}

EOT

ln -s /etc/nginx/sites-available/sonarqube /etc/nginx/sites-enabled/sonarqube

systemctl enable nginx.service

#systemctl restart nginx.service

sudo ufw allow 80,9000,9001/tcp

echo "System reboot in 30 sec"

sleep 30

reboot

# Jenkins Plugins:

Nexus

SonarQube scanner

Git

Pipeline Maven integration plugin

Pipeline utility steps

build Timestamp

# PIPELINE AS A CODE INTRO

Jenkinsfile declares stages in Pipeline

❖ Automate pipeline setup with Jenkinsfile

❖ Jenkinsfile defines Stages in CI/CD Pipeline

❖ Jenkinsfile is a text file with Pipeline DSL Syntax

❖ Similar to groovy

❖ Two Syntax

➢ Scripted

➢ Declarative

## Pipeline Concepts

★ Pipeline

★ Agent

★ Stage

★ Step

pipeline {

agent any

stages {

stage('Build') {

steps {

//

}

}

stage('Test') ) {

steps {

//

}

}

stage('Deploy') { steps { // }

}

}

}

pipeline { ..

…

…..

}

pipeline {

agent { }

tools { }

environment { }

stages { }

}

pipeline {

agent { label "master" }

Tools { maven "Maven" }

}

## EXAMPLE

pipeline {

environment {

NEXUS\_VERSION = "nexus3" NEXUS\_PROTOCOL = "http"

NEXUS\_URL = "you-ip-addr-here:8081" NEXUS\_REPOSITORY = "maven-nexus-repo" NEXUS\_CREDENTIAL\_ID = "nexus-user-credentials" ARTVERSION = "${env.BUILD\_ID}"

TIME = “${BUILD\_TIMESTAMP}”

}

}

pipeline {

stages {

stage("Clone code from VCS") { }

stage("Maven Build") { }

stage("Publish to Nexus Repository Manager") { } } }

pipeline { stages { stage("Clone code from VCS") { steps { } post { } } }

pipeline { stage('BuildAndTest'){ steps { sh 'mvn install' } post { success { echo 'Now Archiving...' archiveArtifacts artifacts: '\*\*/target/\*.war' } } } }

pipeline {

agent any

tools{

maven "MAVEN3"

jdk "OracleJDK17"

}

stages{

stage('Fectch code'){

steps{

git branch: 'main', url: 'https://github.com/hkhcoder/vprofile-project.git'

}

}

stage('build'){

steps{

sh 'mvn install -DskipTests'

}

post {

success {

echo 'archiving artifact now'

//archiveArtifacts artifects: '\*\*/\*.war'

archiveArtifacts artifacts: '\*\*/target/\*.war'

}

}

}

stage('test'){

steps{

sh 'mvn test'

}

}

}

}

## <https://www.jenkins.io/doc/book/pipeline/jenkinsfile/>

# CODE ANALAYSIS

## Why Code Analysis?

● Best Practices

● Vulnerabilities in code

● Functional Errors before deployment

Variety of test performed on the CODE.

Top 10 owas vulnerabilities

## Tools

* Checkstyle
* Cobertura
* mstest
* owasp
* SonarQube Scanner

Etc ..

stage('Checkstyle Analysis'){

steps {

sh 'mvn checkstyle:checkstyle'

}

}

stage('Sonar Analysis') {

environment {

scannerHome = tool 'sonar4.7'

}

steps {

withSonarQubeEnv('sonar') {

sh '''${scannerHome}/bin/sonar-scanner -Dsonar.projectKey=vprofile \

-Dsonar.projectName=vprofile \

-Dsonar.projectVersion=1.0 \

-Dsonar.sources=src/ \

-Dsonar.java.binaries=target/test-classes/com/visualpathit/account/controllerTest/ \

-Dsonar.junit.reportsPath=target/surefire-reports/ \

-Dsonar.jacoco.reportsPath=target/jacoco.exec \

-Dsonar.java.checkstyle.reportPaths=target/checkstyle-result.xml'''

}

}

}

We can search in chatgpt also like “Jenkinsfile to scan the source code with sonar scanner and upload the result to nexus sonarqube server dashboard and with sonar scanner with properties of junt , jacoco & checkstyle report path”

<https://www.jenkins.io/doc/pipeline/steps/sonar/>

# ChatGPT 🡪 sonarqube

sudo mv /opt/nexus/nexus-3.68.1-02 /opt/nexus/nexus

ls -l /opt/nexus/nexus/bin/nexus

[Unit]

Description=Nexus service

After=network.target

[Service]

Type=forking

ExecStart=/opt/nexus/nexus/bin/nexus start

ExecStop=/opt/nexus/nexus/bin/nexus stop

User=nexus

Group=nexus

Restart=on-failure

LimitNOFILE=65536

[Install]

WantedBy=multi-user.target

sudo chown -R nexus:nexus /opt/nexus/nexus

sudo chown -R nexus:nexus /opt/nexus/sonatype-work

sudo systemctl daemon-reload

sudo systemctl start nexus.service

sudo systemctl enable nexus.service

sudo systemctl status nexus.service

# Quality gate

In sonar qube🡪quality gate 🡪we can create our own quality gate(we can add the condition like if the bugs is more then 60th then build should fail)

If the quality gate is running it hold the more time then fail , in that case create the webhooks(sonarqube🡪project setting 🡪webhooks🡪set the url of Jenkins and create a webhook) or sonerqube security group (<http://ip-address> of jenkins/ sonerqube-webhook/

stage("Quality Gate") {

steps {

timeout(time: 1, unit: 'HOURS') {

// Parameter indicates whether to set pipeline to UNSTABLE if Quality Gate fails

// true = set pipeline to UNSTABLE, false = don't

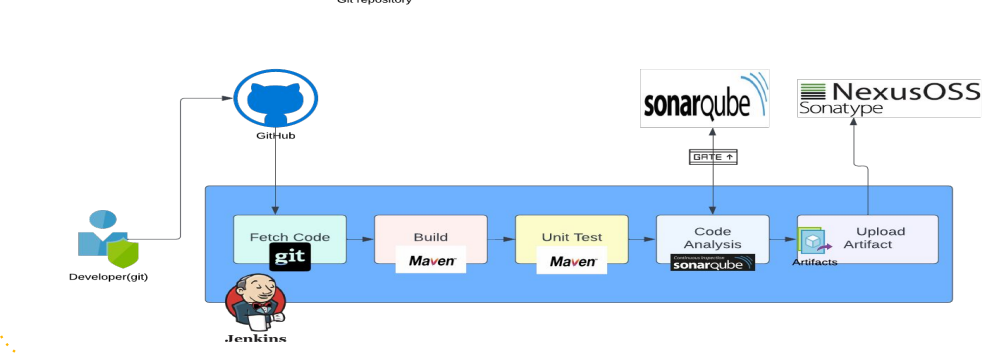
waitForQualityGate abortPipeline: true

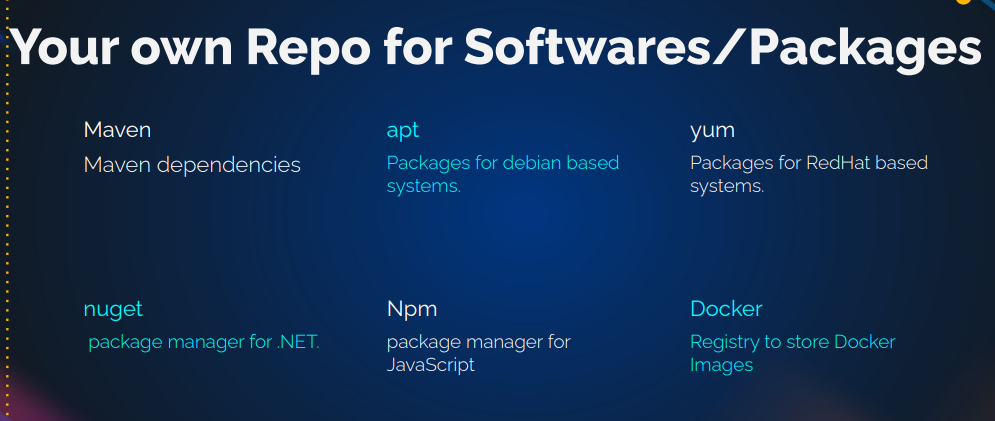
}

}

}

# Software repositories intro (Nexus)





## Nexus Software Repository Manager.

Key Points

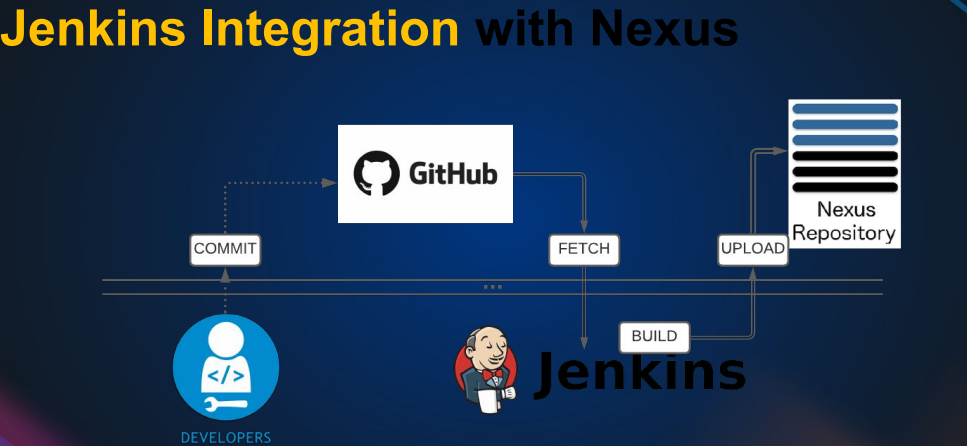
● Runs on java

● Used to store artifacts

● Used as a Package manager for dependencies

● Opensource & Enterprise Versions

● Supports Variety of repo like maven,apt,docker, Ruby gems etc…



One Jenkins build the artifacts and code analysis though sonar scanner then upload in Nexus repo, after that ops team will takes the artifact and deploy in servers

**Login to the nexus repo 🡪 repository 🡪 create repository( host**

Host 🡪store the artifacts

Proxy 🡪 download a dependences

Group 🡪 it is too group the both the repo together(host,proxy)

Create a maven host repo to the reports and artifacts

* Add the nexus credential in Jenkins( dashboard 🡪credentials🡪system🡪global credentials 🡪add the nexus credentials and save it)

stage("UploadArtifact"){

steps{

nexusArtifactUploader(

nexusVersion: 'nexus3',

protocol: 'http',

nexusUrl: '172.31.18.28:8081',

groupId: 'QA',

version: "${env.BUILD\_ID}-${env.BUILD\_TIMESTAMP}",

repository: 'vprofile-repo',

credentialsId: 'nexuslogin',

artifacts: [

[artifactId: 'vproapp',

classifier: '',

file: 'target/vprofile-v2.war',

type: 'war']

]

)

}

}

<https://plugins.jenkins.io/nexus-artifact-uploader/>

Jenkins 🡪manage Jenkins 🡪configure system 🡪 update SonarQube IP to private (no need to change ip once restart)

# NOTEFICATION & SLACKS

def COLOR\_MAP = [

'SUCCESS': 'good',

'FAILURE': 'danger',

]

pipeline {

agent any

tools {

maven "MAVEN3"

jdk "OracleJDK8"

}

stages{

stage('Print error'){

steps{

sh 'fake comment'

}

}

stage('Fetch code') {

steps{

git branch: 'vp-rem', url:'https://github.com/devopshydclub/vprofile-repo.git'

}

}

stage('Build') {

steps {

sh 'mvn clean install -DskipTests'

}

post {

success {

echo "Now Archiving."

archiveArtifacts artifacts: '\*\*/\*.war'

}

}

}

stage('Test'){

steps {

sh 'mvn test'

}

}

stage('Checkstyle Analysis'){

steps {

sh 'mvn checkstyle:checkstyle'

}

}

stage('Sonar Analysis') {

environment {

scannerHome = tool 'sonar4.7'

}

steps {

withSonarQubeEnv('sonar') {

sh '''${scannerHome}/bin/sonar-scanner -Dsonar.projectKey=vprofile \

-Dsonar.projectName=vprofile \

-Dsonar.projectVersion=1.0 \

-Dsonar.sources=src/ \

-Dsonar.java.binaries=target/test-classes/com/visualpathit/account/controllerTest/ \

-Dsonar.junit.reportsPath=target/surefire-reports/ \

-Dsonar.jacoco.reportsPath=target/jacoco.exec \

-Dsonar.java.checkstyle.reportPaths=target/checkstyle-result.xml'''

}

}

}

stage("Quality Gate") {

steps {

timeout(time: 1, unit: 'HOURS') {

// Parameter indicates whether to set pipeline to UNSTABLE if Quality Gate fails

// true = set pipeline to UNSTABLE, false = don't

waitForQualityGate abortPipeline: true

}

}

}

stage("UploadArtifact"){

steps{

nexusArtifactUploader(

nexusVersion: 'nexus3',

protocol: 'http',

nexusUrl: '172.31.18.28:8081',

groupId: 'QA',

version: "${env.BUILD\_ID}-${env.BUILD\_TIMESTAMP}",

repository: 'vprofile-repo',

credentialsId: 'nexuslogin',

artifacts: [

[artifactId: 'vproapp',

classifier: '',

file: 'target/vprofile-v2.war',

type: 'war']

]

)

}

}

}

post {

always {

echo 'Slack Notifications.'

slackSend channel: '#jenkinscicd',

color: COLOR\_MAP[currentBuild.currentResult],

message: "\*${currentBuild.currentResult}:\* Job ${env.JOB\_NAME} build ${env.BUILD\_NUMBER} \n More info at: ${env.BUILD\_URL}"

}

}

}

## Note:

Slack is used for notification (used to integrate with it) 🡪 <https://slack.com/intl/en-in/get-started#/createnew>

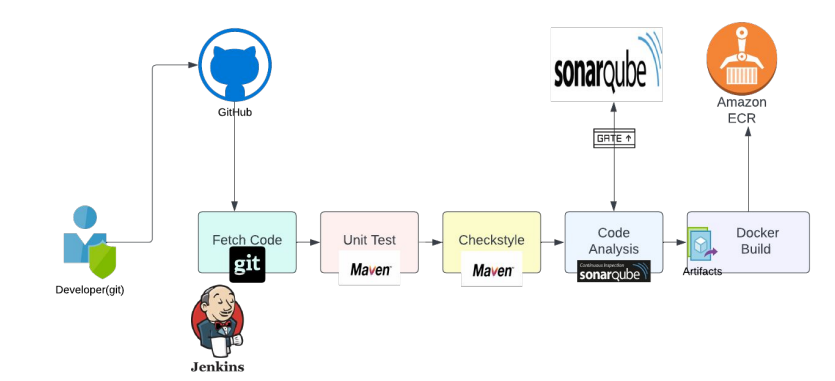
Create a workspace 🡪 name: vprofile 🡪working on devops cicd 🡪create a chennel (it like a group)

To create a token 🡪https://slack.com/apps/A0F7VRFKN-jenkins-ci 🡪add the slack🡪 channel name

Save the token

Jenkins 🡪 install the slack nofication plugin 🡪 add the token to secret text 🡪 give the channel name

# CI for DOCKER



In this we are using docker to containze the artifact

Develops develop the code and commit the github from github Jenkins will detect it fetect the code and will run unit test, code analysis by using checksytyle, sonar qube and upload the result into sonarqube server then we are go into build docker image.

ECR 🡪 Elastic container registry(AWS)

GCR 🡪 Google container registry(google)

Azure registory service

Docker hub

Nexus etc

# Docker PAAC Prereqs info

pipeline {

agent any

tools {

maven "MAVEN3"

jdk "OracleJDK8"

}

environment {

registryCredential = 'ecr:us-east-2:awscreds'

appRegistry = "951401132355.dkr.ecr.us-east-2.amazonaws.com/vprofileappimg"

vprofileRegistry = "https://951401132355.dkr.ecr.us-east-2.amazonaws.com"

}

//

stages {

stage('Fetch code'){

steps {

git branch: 'docker', url: 'https://github.com/devopshydclub/vprofile-project.git'

}

}

stage('Test'){

steps {

sh 'mvn test'

}

}

stage ('CODE ANALYSIS WITH CHECKSTYLE'){

steps {

sh 'mvn checkstyle:checkstyle'

}

post {

success {

echo 'Generated Analysis Result'

}

}

}

stage('build && SonarQube analysis') {

environment {

scannerHome = tool 'sonar4.7'

}

steps {

withSonarQubeEnv('sonar') {

sh '''${scannerHome}/bin/sonar-scanner -Dsonar.projectKey=vprofile \

-Dsonar.projectName=vprofile-repo \

-Dsonar.projectVersion=1.0 \

-Dsonar.sources=src/ \

-Dsonar.java.binaries=target/test-classes/com/visualpathit/account/controllerTest/ \

-Dsonar.junit.reportsPath=target/surefire-reports/ \

-Dsonar.jacoco.reportsPath=target/jacoco.exec \

-Dsonar.java.checkstyle.reportPaths=target/checkstyle-result.xml'''

}

}

}

stage("Quality Gate") {

steps {

timeout(time: 1, unit: 'HOURS') {

// Parameter indicates whether to set pipeline to UNSTABLE if Quality Gate fails

// true = set pipeline to UNSTABLE, false = don't

waitForQualityGate abortPipeline: true

}

}

}

stage('Build App Image') {

steps {

script {

dockerImage = docker.build( appRegistry + ":$BUILD\_NUMBER", "./Docker-files/app/multistage/")

}

}

}

stage('Upload App Image') {

steps{

script {

docker.withRegistry( vprofileRegistry, registryCredential ) {

dockerImage.push("$BUILD\_NUMBER")

dockerImage.push('latest')

}

}

}

}

}

}

IAM user with ECR permissions

Store aws creditentials in Jenkins

ECR repos on AWS

Plugin docker pipeline

ECR Plugin

Install docker engine on jenkins

## Docker ci in Jenkins

Install docker enginer in Jenkins

Add Jenkins user to docker group & reboot

Install AWS CLI

Create IAM USER

Create ECR repo

Plugin 🡪 ECR, Docker pipeline, aws sdk, cloudbee docker build and publish for credentials

In Jenkins server:

Sudo apt update && sudo apt install awscli -y

Install docker engine:

Run the following command to uninstall all conflicting packages:

**for** pkg in docker.io docker-doc docker-compose podman-docker containerd runc**;** **do** sudo apt-get remove $pkg**;** **done**

Set up Docker's Apt repository.

*# Add Docker's official GPG key:*

sudo apt-get update

sudo apt-get install ca-certificates curl gnupg

sudo install -m **0755** -d /etc/apt/keyrings

curl -fsSL https://download.docker.com/linux/ubuntu/gpg **|** sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

sudo chmod a+r /etc/apt/keyrings/docker.gpg

*# Add the repository to Apt sources:*

echo \

"deb [arch="**$(**dpkg --print-architecture**)**" signed-by=/etc/apt/keyrings/docker.gpg]

"**$(**. /etc/os-release **&&** echo "$VERSION\_CODENAME"**)**" stable" **|** \

sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update

Install the Docker packages.

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin

## <https://docs.docker.com/engine/install/ubuntu/>

docker images

su -jenkins

docker images🡪 will get access denied because Jenkins is not on the group of docker

usermod -a -G docker Jenkins

id Jenkins

reboot

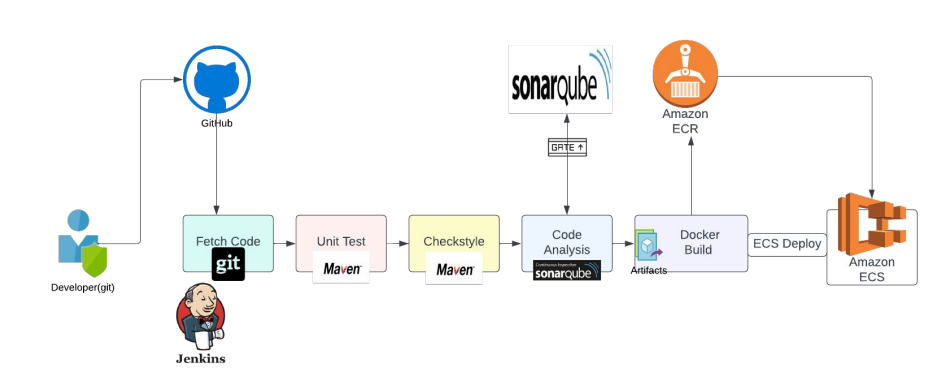
create a AWS IAM user with name – Jenkins: **ECRFULLACCESS**, **AMAZONEC2containerRegistoryFullAcess**

create a access key

Jenkins user🡪 security credentials🡪access key 🡪ceate accesskey🡪cli🡪create key🡪download key

Amazon ECR 🡪 create a repo 🡪 name jenkinsimg

# DOCKER CICD INTRO



We know ci pipeline upto now.

Now we need to host this docker image or the docker or the docker container to a docker solution like ECS.

ECS 🡪 Docker container hosting platform,

We package our images and then we host our dockerized application on ECS.

## Container hosting platform

Docker engine

We can run(docker run image) then your application on the docker engine but it good for testing and local devopment environment.

Kubernetes:

Kubernetes you can use for production environment due to less setting high availability self-healing etc.

-🡪 standalone, EKS,AKS, GKE, Openshift etc

AWS ECS : Elastic container service

pipeline {

agent any

environment {

registryCredential = 'ecr:us-east-2:awscreds'

appRegistry = "951401132355.dkr.ecr.us-east-2.amazonaws.com/vprofileappimg"

vprofileRegistry = "https://951401132355.dkr.ecr.us-east-2.amazonaws.com"

cluster = "vprofile"

service = "vprofileappsvc"

stage('Deploy to ecs') {

steps {

withAWS(credentials: 'awscreds', region: 'us-east-2') {

sh 'aws ecs update-service --cluster ${cluster} --service ${service} --force-new-deployment'

}

}

}

the concepts of deploying Docker images and hosting them using Amazon ECS in a more straightforward way:

1. **Docker Images and Containers:**
   * Imagine a Docker image as a neatly packaged box containing everything your application needs to run: the code, libraries, and dependencies.
   * When you “dockerize” your application, you create these images.
   * Containers are like instances of those images. You can run multiple containers from the same image.
2. **Local Development vs. Production:**
   * During development, you might run containers directly on your local machine using the Docker engine. It’s quick and easy for testing.
   * But for production (when your app is live), you need more robust features like high availability and self-healing.
3. **Enter Amazon ECS (Elastic Container Service):**
   * ECS is like a specialized platform for hosting your Docker containers in the cloud.
   * It’s secure, reliable, and scalable.
   * Think of it as a place where you can put your Docker boxes (images) and let ECS manage them for you.
4. **How ECS Works:**
   * You create Docker images (your neatly packed boxes) and store them in Amazon ECR (Elastic Container Registry).
   * ECS fetches these images and runs containers from them.
   * ECS handles the underlying infrastructure, like servers and networking, so you don’t have to worry about it.
5. **Why Use ECS?**
   * **Reliability:** ECS ensures your containers keep running even if a server fails.
   * **Scalability:** You can easily scale up or down based on demand.
   * **Security:** ECS provides a secure environment for your containers.
   * **Automation:** You can automate deployment using tools like Jenkins (which you mentioned earlier).
6. **Kubernetes (K8s):**
   * Kubernetes is another powerful option for managing containers. It’s like the superhero of container orchestration.
   * We’ll explore Kubernetes in-depth later, but for now, know that it’s great for large-scale deployments.
7. **Next Steps:**
   * In your pipeline, add a stage to deploy your Docker images to ECS.
   * Fulfill the prerequisites (like setting up ECS) to make it work seamlessly.

Remember, ECS simplifies the process of hosting your Dockerized applications in a production-ready environment. If you have any more questions, feel free to ask! 😊

Great, let's look at the detailed steps to create the CICD pipeline using Jenkins to deploy a Docker image to AWS ECS. Here’s how you can set up and configure the pipeline, including the creation of the ECS cluster and service, and the Jenkins pipeline script to automate the deployment.

### Step-by-Step Guide

#### 1. Prerequisites

* **AWS Account**: Ensure you have an AWS account with appropriate permissions to create and manage ECS clusters, services, and IAM roles.
* **Jenkins**: Ensure Jenkins is installed and running. You also need to install the necessary plugins in Jenkins.

#### 2. Install Jenkins Plugins

Install the following Jenkins plugins:

* AWS Steps Plugin: Allows Jenkins to interact with AWS services.
* Docker Pipeline Plugin: Provides Jenkins pipeline capabilities for Docker.

To install the plugins:

1. Navigate to **Manage Jenkins** > **Manage Plugins**.
2. Search for "AWS Steps" and "Docker Pipeline" and install them.

#### 3. Configure AWS Credentials in Jenkins

1. Navigate to **Manage Jenkins** > **Manage Credentials**.
2. Add a new credential with the following details:
   * **Kind**: AWS Credentials.
   * **ID**: aws-credentials.
   * **Access Key ID** and **Secret Access Key**: Your AWS IAM user credentials with the necessary permissions.

#### 4. Create ECS Cluster and Service

Use the AWS Management Console or AWS CLI to create an ECS cluster and a service. Here’s a quick overview using the AWS CLI:

bash

Copy code

# Create ECS cluster

aws ecs create-cluster --cluster-name vprofile

# Create ECS task definition (example configuration)

aws ecs register-task-definition --family vprofile-app-task \

--network-mode awsvpc \

--requires-compatibilities FARGATE \

--cpu "256" \

--memory "512" \

--container-definitions '[{

"name": "vpro-app",

"image": "123456789012.dkr.ecr.us-east-1.amazonaws.com/vprofile-app:latest",

"essential": true,

"portMappings": [{

"containerPort": 8080,

"hostPort": 8080,

"protocol": "tcp"

}],

"logConfiguration": {

"logDriver": "awslogs",

"options": {

"awslogs-group": "/ecs/vprofile",

"awslogs-region": "us-east-1",

"awslogs-stream-prefix": "ecs"

}

}

}]'

# Create ECS service

aws ecs create-service --cluster vprofile \

--service-name vprofile-app-svc \

--task-definition vprofile-app-task \

--desired-count 1 \

--launch-type FARGATE \

--network-configuration '{

"awsvpcConfiguration": {

"subnets": ["subnet-0bb1c79de3EXAMPLE"],

"securityGroups": ["sg-0bb1c79de3EXAMPLE"],

"assignPublicIp": "ENABLED"

}

}'

#### 5. Jenkins Pipeline Script

Create a Jenkins pipeline script (Jenkinsfile) to automate the build and deployment process:

groovy

Copy code

pipeline {

agent any

environment {

AWS\_CREDENTIALS = credentials('aws-credentials')

REGION = 'us-east-1'

CLUSTER\_NAME = 'vprofile'

SERVICE\_NAME = 'vprofile-app-svc'

}

stages {

stage('Build Docker Image') {

steps {

script {

def app = docker.build("vprofile-app")

}

}

}

stage('Push Docker Image to ECR') {

steps {

script {

withDockerRegistry([credentialsId: 'ecr:us-east-1:aws-credentials', url: "https://123456789012.dkr.ecr.us-east-1.amazonaws.com"]) {

def app = docker.build("vprofile-app")

app.push('latest')

}

}

}

}

stage('Deploy to ECS') {

steps {

withAWS(credentials: 'aws-credentials', region: 'us-east-1') {

sh '''

aws ecs update-service --cluster $CLUSTER\_NAME --service $SERVICE\_NAME --force-new-deployment

'''

}

}

}

}

}

#### 6. Explanation of the Jenkins Pipeline

1. **Build Docker Image**:
   * Uses Docker Pipeline plugin to build the Docker image with the application code.
2. **Push Docker Image to ECR**:
   * Authenticates to the AWS ECR registry.
   * Tags and pushes the Docker image to the ECR repository.
3. **Deploy to ECS**:
   * Uses AWS CLI commands to update the ECS service.
   * Forces a new deployment to update the service with the latest Docker image.

#### 7. Triggering the Pipeline

* Commit the Jenkinsfile to your source code repository.
* Configure a Jenkins job to use the Jenkinsfile from your repository.
* Trigger the job manually or set up a webhook to trigger the job on every commit.

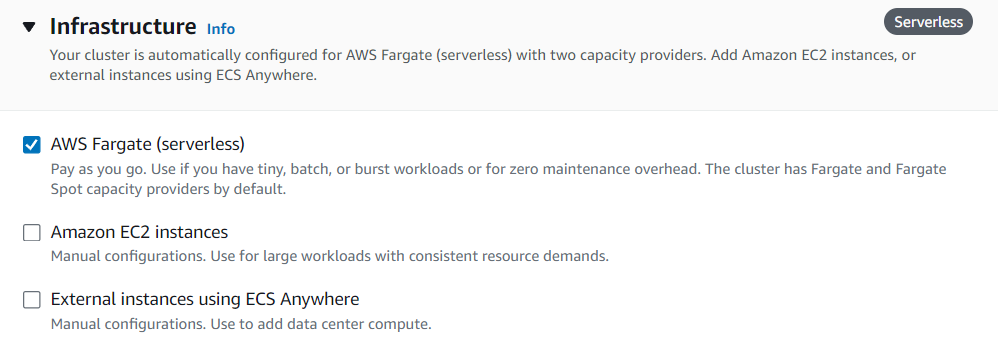
### Summary

1. **Prerequisites**: Ensure AWS account and Jenkins setup.
2. **Install Jenkins Plugins**: Install AWS Steps and Docker Pipeline plugins.
3. **Configure AWS Credentials**: Add AWS credentials in Jenkins.
4. **Create ECS Cluster and Service**: Set up ECS cluster and service using AWS CLI or Management Console.
5. **Jenkins Pipeline Script**: Create a Jenkinsfile to build, push, and deploy the Docker image.
6. **Triggering the Pipeline**: Set up and trigger the Jenkins job.

This setup will automate the entire CI/CD pipeline, allowing you to deploy Docker images to AWS ECS efficiently.

# ECS SETUP

Create a clusters



**Task definition** ( which provides all the information about container)

Provide the cpu, memory

Conatianer info 🡪ECR url and set the port(if service is running in tomcat use 8080 as default)

And check mark(**use log collection)**

Add the tag 🡪name : vprofileapptask

Create a task

In the above task the IAM role is created for default so

Task definition🡪ecs task execution role(in this defaultly you have **amazonecstaskexecutionrolepolicy** is they , add(Attach the roles) 🡪 the other role **cloudwatchlogsfullaccess**

**Create service**

Clusters 🡪service🡪create service

Here we an setup the application type, service type,networking, load balancer and autoscaling etc

And create it

Once service is create check the all the service is up and running like load balancer , health check ,ec2 instance etc

Cicd docker pipeline

pipeline {

agent any

tools {

maven "MAVEN3"

jdk "OracleJDK8"

}

environment {

registryCredential = 'ecr:us-east-2:awscreds'

appRegistry = "951401132355.dkr.ecr.us-east-2.amazonaws.com/vprofileappimg"

vprofileRegistry = "https://951401132355.dkr.ecr.us-east-2.amazonaws.com"

cluster = "vprofile"

service = "vprofileappsvc"

}

stages {

stage('Fetch code'){

steps {

git branch: 'docker', url: 'https://github.com/devopshydclub/vprofile-project.git'

}

}

stage('Test'){

steps {

sh 'mvn test'

}

}

stage ('CODE ANALYSIS WITH CHECKSTYLE'){

steps {

sh 'mvn checkstyle:checkstyle'

}

post {

success {

echo 'Generated Analysis Result'

}

}

}

stage('build && SonarQube analysis') {

environment {

scannerHome = tool 'sonar4.7'

}

steps {

withSonarQubeEnv('sonar') {

sh '''${scannerHome}/bin/sonar-scanner -Dsonar.projectKey=vprofile \

-Dsonar.projectName=vprofile-repo \

-Dsonar.projectVersion=1.0 \

-Dsonar.sources=src/ \

-Dsonar.java.binaries=target/test-classes/com/visualpathit/account/controllerTest/ \

-Dsonar.junit.reportsPath=target/surefire-reports/ \

-Dsonar.jacoco.reportsPath=target/jacoco.exec \

-Dsonar.java.checkstyle.reportPaths=target/checkstyle-result.xml'''

}

}

}

stage("Quality Gate") {

steps {

timeout(time: 1, unit: 'HOURS') {

// Parameter indicates whether to set pipeline to UNSTABLE if Quality Gate fails

// true = set pipeline to UNSTABLE, false = don't

waitForQualityGate abortPipeline: true

}

}

}

stage('Build App Image') {

steps {

script {

dockerImage = docker.build( appRegistry + ":$BUILD\_NUMBER", "./Docker-files/app/multistage/")

}

}

}

stage('Upload App Image') {

steps{

script {

docker.withRegistry( vprofileRegistry, registryCredential ) {

dockerImage.push("$BUILD\_NUMBER")

dockerImage.push('latest')

}

}

}

}

stage('Deploy to ecs') {

steps {

withAWS(credentials: 'awscreds', region: 'us-east-2') {

sh 'aws ecs update-service --cluster ${cluster} --service ${service} --force-new-deployment'

// it going to create new container and old container will delete

// we can see ECS🡪cluster 🡪 vprofile🡪task🡪cleck the task🡪we can see the container

}

}

}

}

}

// **install plugin pipeline:steps**

steps to set up an ECS cluster and run a container using AWS ECS (Elastic Container Service).

### Detailed Step-by-Step Guide

#### 1. Switch to the New ECS Experience

* **Login to AWS Management Console**: Go to the AWS Management Console and navigate to the ECS (Elastic Container Service) dashboard.
* **Enable New ECS Experience**: Ensure the toggle switch for the new ECS experience is turned on. This will provide a more modern and streamlined interface for managing your ECS resources.

#### 2. Create an ECS Cluster

1. **Navigate to Clusters**:
   * Go to the **Clusters** section in the ECS dashboard.
   * Click on **Create Cluster**.
2. **Cluster Configuration**:
   * **Name**: Enter a name for your cluster, such as vprofile.
   * **VPC and Subnets**: Use the default VPC and subnets provided by AWS. This will automatically select all availability zones in your region.
   * **Cluster Configuration**:
     + **Capacity Providers**: Choose **AWS Fargate** to manage the underlying compute resources. This is a serverless option where AWS handles the infrastructure, scaling, and maintenance.
   * **Monitoring**:
     + Enable **Container Insights** to collect metrics and logs, which can be viewed in CloudWatch for monitoring purposes.
   * **Tags**:
     + Click on **Add Tag** and add a tag such as Name: vprofile. Tags are useful for organizing and managing your AWS resources.
3. **Create the Cluster**:
   * Click on **Create** to start the creation process.
   * Wait for the cluster creation to complete. This may take a few minutes.

#### 3. Create a Task Definition

1. **Navigate to Task Definitions**:
   * Go to the **Task Definitions** section in the ECS dashboard.
   * Click on **Create New Task Definition**.
2. **Task Definition Configuration**:
   * **Task Definition Family**: Enter a name for the task definition, such as vprofile-app-task.
   * **Launch Type Compatibility**: Choose **Fargate**.
   * **Operating System Family**: Select **Linux**.
   * **CPU and Memory**:
     + **CPU**: Choose 1 vCPU (the minimum required).
     + **Memory**: Choose 2 GB (the minimum required).
3. **Task Roles**:
   * **Task Role**: Leave it as None initially. This role can be used to give the task permissions to access other AWS services.
   * **Task Execution Role**: Choose to create a new role. This role will be used by ECS to pull images and store logs.
4. **Container Definitions**:
   * **Add Container**:
     + **Name**: Enter a name for the container, such as vpro-app.
     + **Image**: Enter the URI of the Docker image stored in your Amazon ECR, e.g., 123456789012.dkr.ecr.us-east-1.amazonaws.com/vprofile-app:latest.
     + **Port Mappings**: Set the **Container Port** to 8080 (assuming your application runs on port 8080).
   * **Log Configuration**:
     + Enable logging and choose CloudWatch Logs.
     + Configure the log group and stream prefix.
5. **Tags**:
   * Click on **Add Tag** and add a tag such as Name: vprofile-app-task.
6. **Create the Task Definition**:
   * Click on **Create** to save the task definition.

#### 4. Update the IAM Role

1. **Navigate to IAM Roles**:
   * Go to the IAM (Identity and Access Management) service in the AWS Management Console.
   * Find the role created for the ECS task execution (usually named something like ecsTaskExecutionRole).
2. **Attach Policies**:
   * Click on the role to open its details.
   * Click on **Attach Policies** and search for **CloudWatchLogsFullAccess**.
   * Attach the **CloudWatchLogsFullAccess** policy to the role. This allows the ECS tasks to send logs to CloudWatch.

#### 5. Create an ECS Service

1. **Navigate to Your Cluster**:
   * Go back to the **Clusters** section in the ECS dashboard.
   * Select your cluster (e.g., vprofile).
2. **Create a Service**:
   * Go to the **Services** tab and click on **Create**.
3. **Service Configuration**:
   * **Launch Type**: Select **Fargate**.
   * **Task Definition**: Choose the task definition created earlier (e.g., vprofile-app-task).
   * **Service Name**: Enter a name for the service, such as vprofile-app-svc.
   * **Number of Tasks**: Set the desired number of tasks to 1.
4. **Deployment Configuration**:
   * **Deployment Type**: Keep the default rolling update.
   * **Failure Detection**: Uncheck the option for deployment failure detection initially.
5. **Networking Configuration**:
   * **VPC and Subnets**: Use the default VPC and subnets.
   * **Security Groups**:
     + Create a new security group for your service and load balancer.
     + Name it something like vpro-app-ecs-elb-sg.
     + Add rules to allow HTTP traffic on port 80 and custom TCP traffic on port 8080 from anywhere.
6. **Load Balancer Configuration**:
   * **Load Balancer Type**: Choose **Application Load Balancer**.
   * **Load Balancer Name**: Enter a name, e.g., vpro-app-elb-ecs.
   * **Listeners**:
     + Add a listener for port 80.
     + Map it to container port 8080.
   * **Target Group**:
     + Create a new target group, name it something like vpro-ecs-tg.
     + Set the health check path to /login.
7. **Create the Service**:
   * Click on **Create Service** to start the service creation process.
   * This may take a few minutes. The service will create the necessary load balancer, target group, and launch the specified number of tasks (containers).

#### 6. Verify the Service

1. **Check Service Status**:
   * In the ECS dashboard, go to your cluster and select the service you created.
   * Ensure the status is **Active** and the desired number of tasks are running.
2. **Check Task and Load Balancer**:
   * **Tasks**: Go to the **Tasks** tab and verify that the task is running.
   * **Load Balancer**: Go to the EC2 dashboard, select **Load Balancers**, and verify that your load balancer is active.
   * **Target Group**: Check the health status of your target group to ensure that the container is healthy.
3. **Access the Application**:
   * In the ECS service details, go to the **Networking** tab.
   * Copy the **Load Balancer Endpoint** URL.
   * Open a web browser and paste the URL. You should see your application running.
   * Verify that the application is accessible and functioning correctly.

### Summary

1. **Cluster Creation**: Created an ECS cluster using AWS Fargate for serverless container management, enabling easy scaling and minimal maintenance.
2. **Task Definition**: Defined the task with the necessary container details, such as the Docker image URI, CPU, memory, and logging configuration.
3. **IAM Role Update**: Added necessary permissions to the ECS Task Execution Role to allow it to send logs to CloudWatch.
4. **Service Creation**: Set up an ECS service to manage the container, including networking and load balancing configurations.
5. **Verification**: Verified the service, task, and load balancer to ensure everything is running correctly and the application is accessible.

This setup demonstrates how to deploy and manage containerized applications using AWS ECS with a focus on serverless computing using AWS Fargate. In the next lecture, we will automate this deployment process using a CI/CD pipeline.

### Deploying a Docker Container to AWS ECS Using Jenkins

In this session, we will walk through the process of deploying a Docker container to AWS ECS using a Jenkins pipeline. This includes setting up Jenkins, configuring the necessary plugins, and running the pipeline script.

### Step-by-Step Guide

#### Prerequisites

1. **Jenkins Server**: Jenkins installed and running.
2. **AWS CLI**: AWS CLI installed and configured on Jenkins.
3. **AWS Credentials**: Proper AWS credentials set up with permissions for ECS.
4. **Docker**: Docker Engine installed on Jenkins.
5. **AWS ECS Cluster**: An existing ECS cluster and service.

#### Installing AWS Pipeline Plugin

1. **Access Jenkins**: Log into your Jenkins dashboard.
2. **Manage Plugins**:
   * Go to Manage Jenkins > Manage Plugins.
   * In the Available tab, search for Pipeline: AWS Steps.
   * Install the plugin.

#### Create Jenkins Pipeline Job

1. **New Pipeline Job**:
   * From the Jenkins dashboard, click New Item.
   * Enter the name for the job (e.g., CICD Pipeline ECS).
   * Select Pipeline and click OK.
2. **Configure Pipeline Script**:
   * In the job configuration, scroll down to the Pipeline section.
   * Select Pipeline script and enter the following script:

groovy

Copy code

pipeline {

agent any

environment {

AWS\_DEFAULT\_REGION = 'us-east-2' // Change to your region

CLUSTER\_NAME = 'your-cluster-name'

SERVICE\_NAME = 'your-service-name'

}

stages {

stage('Deploy to ECS') {

steps {

withAWS(credentials: 'your-aws-credentials-id', region: "${AWS\_DEFAULT\_REGION}") {

script {

def taskDefinition = sh(script: 'aws ecs describe-services --cluster ${CLUSTER\_NAME} --services ${SERVICE\_NAME} --query "services[0].taskDefinition" --output text', returnStdout: true).trim()

echo "Current Task Definition: ${taskDefinition}"

sh '''

aws ecs update-service \

--cluster ${CLUSTER\_NAME} \

--service ${SERVICE\_NAME} \

--force-new-deployment

'''

}

}

}

}

}

}

* Ensure to replace your-cluster-name, your-service-name, and your-aws-credentials-id with actual values.

#### Verify Deployment

1. **AWS ECS Dashboard**:
   * Go to the AWS ECS console.
   * Check the service under your cluster to see the status of the new deployment.
   * Look for a new task being created and the old task being decommissioned.
2. **Container Logs**:
   * You can verify the logs of the newly created container to ensure the application is running correctly.

#### Summary of the Pipeline Stages

1. **Cluster and Service Information**:
   * Cluster Name: VPN Profile.
   * Service Name: vprofile-app-svc.
2. **AWS CLI Setup**:
   * Ensure AWS CLI is installed and configured on Jenkins.
3. **Plugin Installation**:
   * Install the Pipeline: AWS Steps plugin.
4. **Pipeline Script**:
   * Define the pipeline script to update the ECS service, forcing a new deployment.
5. **Execution and Verification**:
   * Run the pipeline job in Jenkins.
   * Monitor ECS to ensure the new task definition is deployed and the old task is decommissioned.

### Conclusion

By following these steps, you successfully set up a Jenkins pipeline to deploy a Docker container to AWS ECS. This includes fetching the code, building Docker images, uploading them to Amazon ECR, and updating the ECS service to run the new container. This automation ensures a smooth and consistent deployment process, enhancing the CI/CD pipeline for your applications.

### Jenkins Job Triggers: An In-Depth Guide

#### Introduction to Job Triggers

Jenkins jobs can be triggered automatically using various methods, which eliminates the need to manually click the "Build Now" button. Here, we will explore some popular triggers, how they work, and how to set them up.

### Popular Jenkins Triggers

1. **Git Webhooks**
   * **Description**: GitHub (or any Git server) sends a JSON payload to Jenkins whenever there's a commit in the repository.
   * **Usage**: Automatically trigger builds when changes are pushed to the repository.
2. **Poll SCM**
   * **Description**: Jenkins periodically checks the Git repository for new commits at specified intervals.
   * **Usage**: Useful when webhooks are not available. Set intervals like every minute or every five minutes to check for new commits.
3. **Scheduled Jobs**
   * **Description**: Runs jobs at specific times using cron syntax.
   * **Usage**: Automate jobs to run at specific times, e.g., nightly builds or weekly reports.
4. **Remote Triggers**
   * **Description**: Trigger Jenkins jobs from external scripts or systems using an API call.
   * **Usage**: Allows flexibility to trigger builds from various tools such as Ansible playbooks, shell scripts, or other CI/CD tools.
5. **Build After Other Projects Are Built**
   * **Description**: Triggers a job after the completion of another job.
   * **Usage**: Create a pipeline of jobs where the output of one job serves as the input for the next.

### Setting Up Job Triggers

#### Prerequisites

1. **GitHub Repository**: Create a new repository.
2. **SSH Authentication**: Set up SSH keys for secure communication between Jenkins and GitHub.
3. **Jenkins Configuration**: Create a Jenkinsfile, configure global security settings, and set up credentials.

#### Steps to Set Up

1. **Create a GitHub Repository**
   * Log into your GitHub account.
   * Create a new repository (e.g., jenkinstriggers).
2. **Set Up SSH Authentication**
   * **Generate SSH Keys**:

sh

Copy code

ssh-keygen

* + - Follow the prompts to create the keys.
  + **Copy Public Key to GitHub**:
    - Go to Settings > SSH and GPG keys > New SSH Key.
    - Paste the public key (~/.ssh/id\_rsa.pub).

1. **Clone the Repository and Add Jenkinsfile**
   * **Clone Repository**:

sh

Copy code

git clone git@github.com:yourusername/jenkinstriggers.git

* + **Create Jenkinsfile**:

groovy

Copy code

pipeline {

agent any

stages {

stage('Build') {

steps {

echo 'Hello, Jenkins!'

}

}

}

}

* + **Commit and Push Jenkinsfile**:

sh

Copy code

git add Jenkinsfile

git commit -m "Add Jenkinsfile"

git push origin master

1. **Configure Jenkins to Access GitHub Repository**
   * **Global Security Configuration**:
     + Go to Manage Jenkins > Configure Global Security.
     + Under Git Host Key Verification Configuration, select Accept first connection.
     + Save the configuration.
   * **Create a Jenkins Job**:
     + Go to Jenkins Dashboard, click New Item, and create a pipeline job (e.g., build-pipeline).
     + In the job configuration, under Pipeline, select Pipeline script from SCM.
     + Choose Git and enter the repository URL.
     + Add credentials for SSH authentication using SSH username with private key.
2. **Setting Up Job Triggers**
   * **Git Webhook**:
     + Go to your GitHub repository, Settings > Webhooks > Add webhook.
     + URL: http://your-jenkins-url/github-webhook/.
     + Content type: application/json.
     + Trigger: Push events.
   * **Poll SCM**:
     + In the job configuration, under Build Triggers, check Poll SCM.
     + Schedule: H/5 \* \* \* \* (polls every 5 minutes).
   * **Scheduled Jobs**:
     + In the job configuration, under Build Triggers, check Build periodically.
     + Schedule: H 2 \* \* 1-5 (runs at 2 AM on weekdays).
   * **Remote Triggers**:
     + In the job configuration, under Build Triggers, check Trigger builds remotely (e.g., from scripts).
     + Provide an Authentication Token.
     + Use the URL: http://your-jenkins-url/job/build-pipeline/build?token=your-token.
   * **Build After Other Projects Are Built**:
     + In the job configuration, under Build Triggers, check Build after other projects are built.
     + Specify the projects to monitor.

### Testing the Setup

1. **Git Webhook**:
   * Push a commit to the GitHub repository and observe Jenkins triggering the job.
2. **Poll SCM**:
   * Wait for the polling interval and check if the job is triggered on new commits.
3. **Scheduled Jobs**:
   * Verify the job runs at the scheduled time.
4. **Remote Triggers**:
   * Use a script or browser to trigger the job using the provided URL.
5. **Build After Other Projects Are Built**:
   * Complete the specified projects and check if the subsequent job is triggered.

### Conclusion

By setting up various Jenkins triggers, you can automate your CI/CD pipeline effectively, ensuring timely builds and tests. This reduces manual intervention and increases efficiency in the software development lifecycle.

# **Job Triggers**

1. Git Webhook

In github/git repository will send JSON playload whenever there is a commit in the repository.

1. Poll SCM

It is opposite to git webhook, in poll SCM Jenkins will check the commit in git repository. In an interval that you Specified.(like every 5 minutes or minutes or hour)

1. Scheduled jobs

You mention date and time like a alarm clock in crown job format and Jenkins will make sure the job will run in particular time or intervels.

1. Remote triggers

You can trigger Jenkins jobs from anywhere, from ascript, from a ansible playbook, from anywhere for that matter

1. Build after other projects are build

🡪after one job completed run the specified job

Step to do:

1. Create git repository on github 🡪 Jenkinstriggers
2. Ssh auth 🡪 ssh-keygen.exc
3. Create a jenkinsfile in git repo & commit
4. Create Jenkins job to access jenkinsfile from git repo
5. Test triggers

Create a git repository on github(jenkinstriggers) and

Generate the ssh key (ssh-keygen.exe)🡪it will create a public and private key 🡪 copy the public key and 🡪jenkins-setting🡪ssh and gpg key🡪add the key

Create a folder🡪intized git(git init)🡪git add . 🡪 commit the code or clone the repo

Create a Jenkinsfile and place the git repo 🡪 from they we can trigger the file by using webhooks, poll SCM, Schedules etc

If you getting “host key verification failed error”

🡪jenkins🡪manage Jenkins🡪security 🡪 git host key verification 🡪set accept first connection

Create a pipeline🡪pipeline script from SCM 🡪use git 🡪 add url and private key 🡪 add the Jenkinsfile path🡪run the job

pipeline{

agent any

stages{

stage('build'){

steps{

echo "this this jenkins build"

}

}

}

}

Webhooks:-

Copy Jenkins url 🡪go to github setting 🡪Webhooks 🡪 payload URL (<http://localhost:8080/github-webhook/> ) 🡪 content type ( application/json) 🡪 add webhook

In jenkinstrigger folder 🡪 (create folder🡪add , commit & push) 🡪 now job will run

Poll SCM :

It is smillar 🡪 min hour dom(1-31) month dow(1-7)

Scheduled jobs

Build periodly (simllar to Poll SCM)

Remote job:

Usr url , token, crum 🡪 combine all create a curl command ( follow the document devops learning)

### Setting Up a Git Webhook with Jenkins

**Objective**: Automatically trigger a Jenkins job whenever there is a new commit or other specified events in a GitHub repository.

#### Steps and Example:

1. **Copy Jenkins URL**: Assume your Jenkins server URL is http://your-jenkins-url:8080.
2. **Go to GitHub Repository Settings**:
   * Navigate to your GitHub repository, e.g., https://github.com/yourusername/your-repository.
   * Click on Settings in the repository menu.
3. **Add a Webhook**:
   * Go to the Webhooks section and click Add webhook.
   * Paste your Jenkins URL followed by /github-webhook/, so it should look like http://your-jenkins-url:8080/github-webhook/.
   * Set the content type to application/json.
   * Choose the events to trigger the webhook, typically push events.
4. **Verify Webhook**:
   * Once added, a green check mark indicates successful delivery. If you see a red cross, double-check the URL, content type, and ensure Jenkins can be accessed on port 8080.
5. **Configure Jenkins Job**:
   * In Jenkins, go to your job configuration, e.g., http://your-jenkins-url:8080/job/your-job/configure.
   * Under Build Triggers, select GitHub hook trigger for GITScm polling.
   * Save the configuration.
6. **Test the Webhook**:
   * Make a commit in your GitHub repository

git add .

git commit -m "Test commit to trigger Jenkins job"

git push origin master

* + This should trigger the Jenkins job automatically.

### Poll SCM in Jenkins

**Objective**: Jenkins will regularly check for new commits in your GitHub repository and trigger the job if it finds any.

#### Steps and Example:

1. **Configuration**:
   * In the job configuration, select Poll SCM.
   * Define the polling schedule using cron syntax. For example, to poll every minute, use:

\* \* \* \* \*

**Verification**:

* Jenkins will log its polling activity. If it finds new commits, it will trigger the job.
* Make a commit to test

git add .

git commit -m "Test commit for Poll SCM"

git push origin master

### Build Periodically

**Objective**: Run the Jenkins job at specific intervals, regardless of any GitHub activity.

#### Steps and Example:

1. **Configuration**:
   * In the job configuration, select Build periodically.
   * Use cron syntax to define the schedule. For example, to run the job at 8:30 PM every weekday, use:

30 20 \* \* 1-5

### Remote Trigger

**Objective**: Trigger the Jenkins job from an external source, like another script or system.

#### Steps and Example:

1. **Create a Trigger Token**:
   * In the job configuration, enable Trigger builds remotely and set a token, e.g., mybuildtoken.
2. **Generate User API Token**:
   * Go to Jenkins user settings and generate an API token:
     + Jenkins URL: http://your-jenkins-url:8080/user/your-username/configure
     + Click on Add new Token and generate a token, e.g., abcd1234.
3. **Generate CRUMB**:
   * Run a command to generate a CRUMB. In a terminal, use:

sh

wget --auth-no-challenge --user=admin --password=your-password [http://your-jenkins-url:8080/crumbIssuer/api/xml?xpath=concat(//crumbRequestField,":",//crumb)](http://your-jenkins-url:8080/crumbIssuer/api/xml?xpath=concat(//crumbRequestField,%22:%22,//crumb))

**Construct cURL Command**:

* Use the API token and CRUMB in a cURL command to trigger the job remotely:

curl -u admin:abcd1234 -H "$(wget --auth-no-challenge --user=admin --password=your-password http://your-jenkins-url:8080/crumbIssuer/api/xml?xpath=concat(//crumbRequestField,":",//crumb) -q -O -)" <http://your-jenkins-url:8080/job/your-job/build?token=mybuildtoken>

1. **Run the cURL Command**:
   * Paste and run the above cURL command in your terminal. This will trigger the Jenkins job.

### Build After Other Projects

**Objective**: Automatically trigger one job after another job completes.

#### Steps and Example:

1. **Create a New Job**:
   * Create a new job, e.g., test-job, and configure it:
     + Add a build step to execute a shell command

echo "Test, test, test"

1. **Set Up Trigger**:
   * In the test-job configuration, under Build Triggers, select Build after other projects are completed and specify the job it depends on, e.g., build-job.
2. **Test the Setup**:
   * Run the initial job (build-job), and once it completes, it should trigger the test-job.

#### Example:

* Create and configure the build-job.
* Create and configure the test-job:
  + Go to Build Triggers and select Build after other projects are completed.
  + Enter build-job in the project field.
* Run build-job and observe that it triggers test-job upon completion.

### Summary

These examples provide a step-by-step guide to setting up different triggers for Jenkins jobs, automating your build and deployment processes based on various events and conditions.

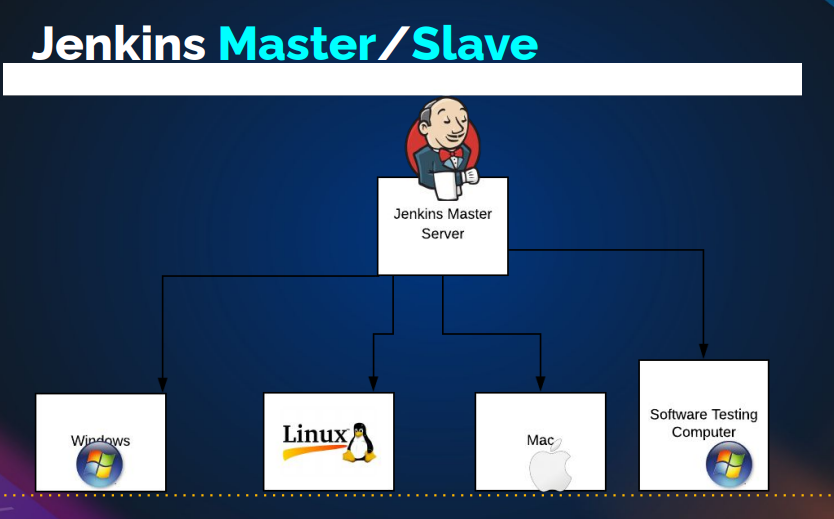
4o

# Jenkins Master & Note

Distributed Builds, Cross Platform builds and much more

Use Cases

1. Load Distribution
2. Jenkins Master Executes Build Job on Node it selected.
3. Cross Platform Builds Executing Build of other platforms like .net(Windows), IOS(Mac OS) from Jenkins Master(Linux)
4. Software Testing Execute Testers Test Automation Scripts from Node.



Execute Anything

Can add any computer in network as Jenkins Node and execute commands on Nodes. e:g scripts, commands, test scripts, etc

Prerequisites for Node Setup

Any OS

2. Network access from Master Note: Check Firewall rules

3. Java, JRE, JDK

4. User

5. Directory with User ownership

6. Tools as required by the Jenkins job e:g Maven, Ant, Git etc

### Jenkins Master/Slave Concepts Explained

#### Introduction to Jenkins Master/Slave Architecture

* **Jenkins Master**: The central server that manages Jenkins jobs, provides the web interface, schedules builds, and delegates tasks.
* **Jenkins Slave (Agent)**: Machines that run build tasks dispatched by the Master. Slaves can run different environments and tools necessary for the build jobs.

#### Why Use Slaves?

1. **Load Distribution**: To distribute the workload across multiple machines, ensuring that the Jenkins Master isn't overloaded.
2. **Cross-Platform Builds**: To build and test software on different operating systems (e.g., Windows, Linux, macOS).
3. **Specific Task Execution**: To run specific scripts, commands, or tests on designated machines with the necessary tools and environment.

#### Setting Up a Slave Machine

##### Prerequisites

1. **Operating System**: Any (Windows, Linux, macOS).
2. **Network Access**: Ensure network connectivity between the Jenkins Master and Slave. Check for firewalls or network restrictions.
3. **Java Installation**: Install Java Runtime Environment (JRE) or Java Development Kit (JDK) as needed.
4. **User Creation**: Create a user on the slave machine that Jenkins will use to connect.
5. **Directory Setup**: Ensure there is a directory that the Jenkins user can access and use for build jobs.
6. **Required Tools**: Install necessary tools like Git, Maven, etc.

##### Example: Adding a Linux Slave (Agent)

**Step 1: Launch an EC2 Instance**

1. Launch a Linux EC2 instance (e.g., Ubuntu 18.04) on AWS.
2. Name it (e.g., Slave-Jenkins).
3. Create and configure a security group (e.g., Slave-SG) to allow SSH (port 22) from the Jenkins Master.

**Step 2: Configure the Linux Slave**

1. **SSH into the Instance**:
   * Use the SSH key (e.g., jenkins-key.pem) to connect.
   * Command: ssh -i jenkins-key.pem ubuntu@<ec2-instance-public-ip>
2. **Install Java**:

sudo apt update

sudo apt install openjdk-11-jdk -y

**Create a User and Directory**:

sudo adduser devops

sudo mkdir /opt/jenkins-slave

sudo chown devops:devops /opt/jenkins-slave

**Enable SSH Password Authentication**:

* Edit the SSH configuration file

sudo nano /etc/ssh/sshd\_config

 Find the line PasswordAuthentication no and change it to PasswordAuthentication yes.

 Save and exit the editor (Ctrl+O, Enter, Ctrl+X).

 Restart SSH service

sudo systemctl restart ssh

**Step 3: Configure Jenkins Master to Add Slave**

1. **Go to Jenkins Dashboard**:
   * Navigate to Manage Jenkins > Manage Nodes and Clouds.
   * Click New Node.
2. **Create New Node**:
   * Name: silver-node
   * Type: Permanent Agent
   * Number of Executors: 1 (number of jobs this node can run in parallel)
   * Remote Root Directory: /opt/jenkins-slave
   * Labels: silver
   * Usage: Use this node as much as possible
   * Launch Method: Launch agent by connecting it to the master using SSH
   * Host: <ec2-instance-private-ip>
3. **Add Credentials**:
   * **Username with Password**: Use the devops user and its password.
   * **SSH Username with Private Key**: (Optional) If you prefer using key-based authentication, provide the private key.
4. **Set SSH Key Verification Strategy**:
   * Choose Non verifying Verification Strategy.
5. **Save and Test Connection**:
   * Click Save.
   * Check the node’s status and logs to ensure it is connected and online.

**Step 4: Run a Job on the Slave**

1. **Create a New Job**:
   * Go to New Item and create a job (e.g., test-slave-job).
   * In the job configuration, add shell commands to run

pwd

whoami

ls -ltr

1. **Restrict Job to Run on Slave**:
   * In the job configuration, check Restrict where this project can be run and specify the label silver.
2. **Build the Job**:
   * Click Build Now.
   * Check the console output to ensure it runs on the silver-node.

**Example: Adding a Windows Slave**

**Step 1: Launch a Windows EC2 Instance**

1. Launch a Windows EC2 instance (e.g., Windows Server 2019).
2. Name it (e.g., Slave-Windows).
3. Create and configure a security group (e.g., Slave-SG) to allow RDP (port 3389) from your IP.

**Step 2: Configure the Windows Slave**

1. **RDP into the Instance**:
   * Use RDP to connect to the instance.
   * Download the RDP file from the AWS console and use it to connect.
2. **Install Java**:
   * Download and install the latest JDK from the [Oracle website](https://www.oracle.com/java/technologies/javase-jdk11-downloads.html).
3. **Create a User**:
   * Create a user (e.g., devops) with administrative privileges.

**Step 3: Configure Jenkins Master to Add Slave**

1. **Go to Jenkins Dashboard**:
   * Navigate to Manage Jenkins > Manage Nodes and Clouds.
   * Click New Node.
2. **Create New Node**:
   * Name: gold-node
   * Type: Permanent Agent
   * Number of Executors: 1
   * Remote Root Directory: C:\jenkins
   * Labels: gold
   * Usage: Use this node as much as possible
   * Launch Method: Launch agent via Java Web Start
3. **Save and Download Agent**:
   * Save the configuration.
   * From the Jenkins node page, download the agent JAR file (e.g., agent.jar).
4. **Run the Agent on Windows Slave**:
   * Transfer agent.jar to the Windows machine.
   * Open Command Prompt and run:

sh

Copy code

java -jar agent.jar -jnlpUrl <Jenkins master URL>/computer/gold-node/slave-agent.jnlp -secret <secret key>

* + This command connects the Windows slave to the Jenkins master.

1. **Verify Connection**:
   * Check the node status in Jenkins to ensure it is connected.

**Step 4: Run a Job on the Windows Slave**

1. **Create a New Job**:
   * Go to New Item and create a job (e.g., test-windows-job).
   * In the job configuration, add batch commands to run

echo %CD%

whoami

dir

1. **Restrict Job to Run on Slave**:
   * In the job configuration, check Restrict where this project can be run and specify the label gold.
2. **Build the Job**:
   * Click Build Now.
   * Check the console output to ensure it runs on the gold-node.

### Conclusion

By setting up Jenkins slaves, you can efficiently distribute build tasks, utilize different environments, and enhance the overall capability and flexibility of your CI/CD pipeline. Properly configuring network settings, user permissions, and necessary tools ensures smooth operation and successful job execution across all nodes.

# Authentication & Authorization

Authentication 🡪 login

Authorization 🡪 privilege

Once the pipeline is created, need to hand it over to developer(so don’t share the login detail , don’t full access)

## Securing Jenkins

1. User Login
2. Jenkins own database.
3. Sign Up
4. LDAP Integration

### Permissions on Jenkins

● Admin

● Read

● Jobs

● Credentials

● Plugins etc

### Permissions on Jobs

● View

● Build

● Delete

● Configure

● etc

### Understanding Jenkins Security: Authentication and Authorization

Welcome to the Jenkins Security session! Here, we'll break down the concepts of authentication and authorization in Jenkins in a simpler way.

#### **Authentication (Who can log in?)**

**Authentication** is all about **logging in** to Jenkins. Right now, we are using an admin user to do everything in Jenkins. When you create a pipeline and hand it over to different teams (developers, testers, operations, etc.), you need to decide how much access or control each team should have.

**Key Points about Authentication:**

1. **Admin User**: This user has full access to everything in Jenkins.
2. **Sign-up**: Users can sign up themselves on Jenkins if allowed.
3. **LDAP Integration**: Jenkins can be connected to your company’s LDAP (like Active Directory), allowing employees to log in with their usual company credentials.

#### **Authorization (What can they do?)**

**Authorization** is about **what users can do** once they are logged in. This determines their permissions and access levels.

**Key Points about Authorization:**

1. **Full Access**: You don’t want non-admin users to have admin-level privileges.
2. **Role-Based Access**: You can fine-tune permissions by creating roles and assigning them to users or groups.
3. **Project-Based Access**: Control access at the project or job level so that users from one project can’t access another project’s resources.

#### **Implementing Authentication and Authorization in Jenkins**

1. **Configuring Authentication**:
   * Go to **Manage Jenkins** and then **Configure Global Security**.
   * Choose how users can log in (authentication). Options include:
     + **Jenkins own user database**: Users sign up directly on Jenkins.
     + **LDAP**: Connects Jenkins to an LDAP server for centralized login.
     + **Operating System Users and Groups**: Not recommended as it mixes system users with Jenkins users.
2. **Configuring Authorization**:
   * **Matrix-based Security**: Fine-tunes permissions at a global level. For example, you can give users specific permissions like build, read, or configure jobs.
   * **Project-based Authorization**: More secure, as it allows you to specify permissions for each job. Users only get access to the jobs they are authorized to work on.
   * **Role-based Authorization**: Use a plugin (Role-based Authorization Strategy) to create roles with specific privileges and assign these roles to users.

#### **Example Setup**

1. **Matrix-based Security**:
   * Add a user and give specific permissions (like build, read, configure) at a global level.
   * This might not be ideal for large organizations with multiple projects as it could grant more access than intended.
2. **Project-based Authorization**:
   * Enable project-based security for specific jobs.
   * Assign permissions to users at the job level, ensuring they can only access and modify the jobs they are supposed to.
3. **Role-based Authorization**:
   * Install the **Role-based Authorization Strategy** plugin.
   * Create roles (like DevOps) with specific permissions.
   * Assign these roles to users, making management easier, especially with many users.

#### **Conclusion**

In summary, Jenkins allows you to control who can log in (authentication) and what they can do (authorization). For better security, especially in larger organizations, use project-based or role-based authorization to ensure users have the appropriate level of access and no more. This setup helps maintain a secure and well-organized Jenkins environment.

### 1. Matrix-based Security

**Path:**

* **Manage Jenkins** > **Configure Global Security** > **Authorization** > **Matrix-based security**

**Steps:**

1. **Go to Manage Jenkins**:
   * From the Jenkins dashboard, click on "Manage Jenkins".
2. **Configure Global Security**:
   * Click on "Configure Global Security".
3. **Select Matrix-based security**:
   * In the Authorization section, select "Matrix-based security".
4. **Add Users and Set Permissions**:
   * Add users by entering their username and set specific permissions like build, read, configure, etc.

### 2. Project-based Authorization Strategy

**Path:**

* **Manage Jenkins** > **Configure Global Security** > **Authorization** > **Project-based Matrix Authorization Strategy**

**Steps:**

1. **Go to Manage Jenkins**:
   * From the Jenkins dashboard, click on "Manage Jenkins".
2. **Configure Global Security**:
   * Click on "Configure Global Security".
3. **Select Project-based Matrix Authorization Strategy**:
   * In the Authorization section, select "Project-based Matrix Authorization Strategy".
4. **Add Users and Set Permissions Globally**:
   * Add users by entering their username and give at least global read access.
5. **Configure Job-specific Permissions**:
   * For each job, go to the job’s configuration page.
   * Check "Enable project-based security".
   * Add users and set job-specific permissions.

### 3. Role-based Authorization Strategy

**Path:**

* **Manage Jenkins** > **Manage Plugins** > **Available** > **Role-based Authorization Strategy** (install the plugin)
* **Manage Jenkins** > **Configure Global Security** > **Authorization** > **Role-based strategy**
* **Manage Jenkins** > **Manage and Assign Roles** > **Manage Roles** / **Assign Roles**

**Steps:**

1. **Install Role-based Authorization Strategy Plugin**:
   * Go to "Manage Jenkins".
   * Click on "Manage Plugins".
   * Go to the "Available" tab.
   * Search for "Role-based Authorization Strategy".
   * Install the plugin and restart Jenkins if required.
2. **Configure Global Security**:
   * After the plugin is installed, go to "Manage Jenkins".
   * Click on "Configure Global Security".
   * In the Authorization section, select "Role-based strategy".
   * Save the configuration.
3. **Manage and Assign Roles**:
   * Go back to "Manage Jenkins".
   * Click on "Manage and Assign Roles".
   * **Manage Roles**:
     + Define roles (e.g., DevOps) and set specific permissions for each role.
   * **Assign Roles**:
     + Assign users to the roles you created.

These paths and steps should help you configure different types of security settings in Jenkins to control both authentication and authorization effectively.