**Jenkins Installation on Ubuntu   
  
STEP\_1: Create a EC2 Machine in AWS (t2.micro)**

SSH to EC2 Machine created above -> sudp apt update -y

**STEP\_2: Install Openjdk Java 17**

 sudo apt install -y openjdk-17-jdk

**STEP\_3: Install Jenkins with bellow commands**

sudo wget -O /usr/share/keyrings/jenkins-keyring.asc https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key  
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 -y jenkins

**STEP\_4: Start Jenkins application suing systemctl**

To start Jenkins service  
      sudo systemctl start jenkins

To stop Jenkins service   
      sudo systemctl stop jenkins

To check current status Jenkins service   
      sudo systemctl status jenkins

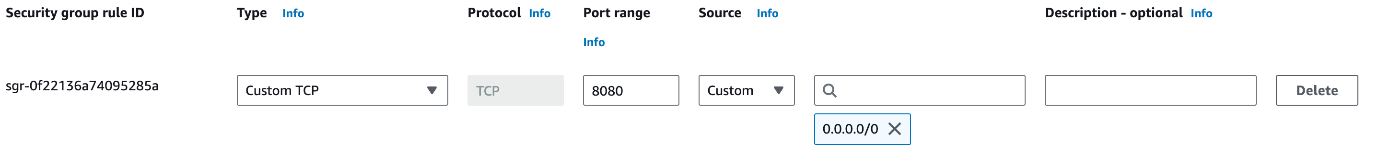
To edit current Jenkins service settings  
      sudo systemctl edit jenkins

**STEP\_5: To access Jenkins from client (Browser)**

Jenkins by default runs on 8080 port  
To open 8080 port to access   
     Goto -> EC2 -> jenkins\_instance -> Security (Tab)

  
Then  
A blue text on a white background

Description automatically generated  
Edit inbound rules  
  
     A white rectangular object with a black line

Description automatically generated with medium confidence  
  
Add inbound rule for 8080   
  
       
Access the jenkins with url <jenkins\_ec2\_public\_ip\_address>:<port> from any browser  
        A close up of a text

Description automatically generated  example: 13.233.131.159:8080

**STEP\_6: Initaial setup of jenkins (Unlock Jenkins)**

**A screenshot of a password

Description automatically generated**

**Run the below command in jenins EC2 machine**         
       sudo cat /var/lib/jenkins/secrets/initialAdminPassword  
         (Note: copy and paste printed key in the above windows)

A close-up of a sign

Description automatically generated  
                       If you get above error means you copied some extra charactes with initial\_admin\_password in the terminal (like including new line)  
                      **Solution:  copy the password into any notepad application and copy only the password to jenkins  
A screenshot of a computer program

Description automatically generated**

**Add Users to jenkins**

Navigate to **Jenkins Dashboard** → **Manage Jenkins** → **Users** → **Create User**. Enter the user name, password, full name and email-address and then click **Create User** button.

**Project-based Matrix Authorization Strategy (User security and access control)**

Project-based Matrix Authorization Strategy allows you to define specific permissions for each project (job) within your Jenkins instance. With this strategy, you can specify which users or groups have particular permissions (such as build, configure, or delete) for individual projects.

Here's how you can set up Project-based Matrix Authorization in Jenkins:

* **Access Jenkins Configuration**: Log in to Jenkins with administrative privileges.
* **Navigate to Manage Jenkins**: Click on the "Manage Jenkins" link on the Jenkins dashboard.
* **Configure Global Security**: Under the "Configure Global Security" section, select the "Enable security" checkbox to enable security settings.
* **Select Project-based Matrix Authorization Strategy**: Scroll down to the "Authorization" section and choose "Project-based Matrix Authorization Strategy" from the available options.
* **Define Permissions**: You will see a matrix where you can define permissions for different users or groups. Here are some common permissions:
  + Overall: Permissions for overall Jenkins access.
  + Job: Permissions related to specific jobs or projects.
  + Run: Permissions to run jobs.
  + Configure: Permissions to configure jobs.
  + Read: Permissions to view job configurations and build results.
  + Workspace: Permissions to access the workspace of a job.
  + Cancel: Permissions to cancel builds.
  + Delete: Permissions to delete jobs.
* **Grant Permissions**: For each permission, you can specify which users or groups should have that permission by checking the corresponding checkboxes.
* **Save Configuration**: Once you have configured the permissions as per your requirements, click on the "Save" or "Apply" button to save the changes.
* **Test Permissions**: After configuring the permissions, it's essential to test them to ensure that users and groups have the appropriate access to Jenkins projects.

**Jenkins Freestyle Project/Job**

**General Configuration:**

* **Job Name:** Provide a meaningful name for the job to identify its purpose.
* **Description:** Optionally, add a description to provide additional context about the job's purpose.
* **Discard Old Builds:** Specifies the strategy for managing old builds to free up disk space  
  Options typically include keeping all builds, keeping only the last N builds, or keeping builds based on days.
* **Execute concurrent builds if necessary:**Allows Jenkins to execute multiple builds of this job concurrently if necessary.
* **This project is parameterized:** Allows you to define parameters for the job, enabling users to customize build behavior.  
  + Most used parameter types,  
    **String Parameter:**Accepts a single string value.  
    **Boolean Parameter:** Accepts a true/false value.  
    **Choice Parameter:**Presents a dropdown menu or radio buttons to select one option from a predefined list.

**Source Code Management (SCM):**

* **SCM:**Choose the version control system (e.g., Git) for your project.
* **Repository URL:** Provide the URL of your version control repository.
* **Credentials:** If required, provide credentials to authenticate with the version control system.

**Build Triggers:**

Specify conditions that trigger a build.

* **Build periodically:** Trigger builds based on a schedule (using cron syntax).
* **Poll SCM**: Trigger a build when changes are detected in the version control repository.
* **Build after other projects are built:** Trigger this job after other specified projects are built.
* **Trigger builds remotely:** Trigger builds remotely via HTTP.

**Build Environment:**

* **Build Environment:** Set up the environment for the build process.
* **Delete workspace before build starts:**Clean up the workspace before starting a new build.
* **Set Environment Variables:** Define environment variables for the build process.

**Build:**

* **Build:**Define the build steps for the job.
* **Execute shell:** Execute shell commands on Unix-like platforms.
* **Execute Windows batch command:** Execute batch commands on Windows.
* **Invoke Maven:** Execute Maven goals.

**Post-Build Actions:**

* **Post-Build Actions:** Define actions to be performed after the build completes.
* **Archive the artifacts:** Archive files (e.g., JAR, WAR) to keep them for future reference.
* **Email Notification:** Send email notifications to specified recipients after the build.
* **Trigger parameterized build:** Trigger another project with parameters.
* **Publish JUnit test result report:** Publish JUnit test results to Jenkins.

**Crontab (Job Scheduler)**

The crontab is a list of commands that you want to run on a regular schedule, and also the name of the command used to manage that list. Crontab stands for “cron table, ” because it uses the job scheduler cron to execute tasks; cron itself is named after “chronos, ” the Greek word for time.cron is the system process which will automatically perform tasks for you according to a set schedule. The schedule is called the crontab, which is also the name of the program used to edit that schedule.

Linux Crontab Format:     \*         \*            \*          \*         \*              <command to execute>

       MIN    HOUR     DOM     MON     DOW                  CMD

FIELD DESCRIPTION ALLOWED VALUE

* MIN Minute field 0 to 59
* HOUR Hour field 0 to 23
* DOM Day of Month 1-31
* MON Month field 1-12
* DOW Day Of Week 0-6
* CMD Command Any command to be executed.

To view the Crontab entries: crontab -l

To edit Crontab Entries: crontab -e

To edit crontab entries of other Linux users: crontab -u username -e

example

1) To schedule a job for every minute using Cron

\* \* \* \* \* command/script

2) How to Execute a Linux Cron Jobs Every Second Using Crontab.

You cannot schedule an every-second cronjob. Because in cron the minimum unit you can specify is minute. 

3) To schedule a background Cron job for every 10 minutes.

\*/10 \* \* \* \* /home/maverick/check-disk-space

4) Schedule a Job for More Than One Instance (e.g. Twice a Day)

executes the specified script at 11:00 and 16:00 every day

00 11,16 \* \* \* /home/ramesh/bin/incremental-backup

5) Schedule a Job for Specific Range of Time (e.g. Only on Weekdays)

    This example checks the status of the database every day (including weekends) during the working hours 9 a.m – 6 p.m

00 09-18 \* \* \* /home/ramesh/bin/check-db-status

6) Cron job to run on the last day of the month

55 23 28-31 \* \* [[ "$(date --date=tomorrow +\%d)" == "01" ]] && myscript.sh 0 23 28-31 \* \* [ $(date -d +1day +%d) -eq 1 ] && myscript.sh

7) Cron special keywords and their meaning Keyword Equivalent

@yearly 0 0 1 1 \*

@daily 0 0 \* \* \*

@hourly 0 \* \* \* \*

@reboot Run at startup.

a) To schedule a job for the first minute of every year using @yearly

@yearly /home/maverick/bin/annual-maintenance

b) To schedule a Cron job beginning of every month using @monthly

@monthly /home/maverick/bin/tape-backup

c) To schedule a background job every day using @daily

@daily /home/maverick/bin/cleanup-logs "day started"

d) To execute a Linux command after every reboot using @reboot

@reboot CMD

**Add Users to jenkins**

Navigate to **Jenkins Dashboard** → **Manage Jenkins** → **Users** → **Create User**. Enter the user name, password, full name and email-address and then click **Create User** button.

**Project-based Matrix Authorization Strategy (User security and access control)**

Project-based Matrix Authorization Strategy allows you to define specific permissions for each project (job) within your Jenkins instance. With this strategy, you can specify which users or groups have particular permissions (such as build, configure, or delete) for individual projects.

Here's how you can set up Project-based Matrix Authorization in Jenkins:

* **Access Jenkins Configuration**: Log in to Jenkins with administrative privileges.
* **Navigate to Manage Jenkins**: Click on the "Manage Jenkins" link on the Jenkins dashboard.
* **Configure Global Security**: Under the "Configure Global Security" section, select the "Enable security" checkbox to enable security settings.
* **Select Project-based Matrix Authorization Strategy**: Scroll down to the "Authorization" section and choose "Project-based Matrix Authorization Strategy" from the available options.
* **Define Permissions**: You will see a matrix where you can define permissions for different users or groups. Here are some common permissions:
  + Overall: Permissions for overall Jenkins access.
  + Job: Permissions related to specific jobs or projects.
  + Run: Permissions to run jobs.
  + Configure: Permissions to configure jobs.
  + Read: Permissions to view job configurations and build results.
  + Workspace: Permissions to access the workspace of a job.
  + Cancel: Permissions to cancel builds.
  + Delete: Permissions to delete jobs.
* **Grant Permissions**: For each permission, you can specify which users or groups should have that permission by checking the corresponding checkboxes.
* **Save Configuration**: Once you have configured the permissions as per your requirements, click on the "Save" or "Apply" button to save the changes.
* **Test Permissions**: After configuring the permissions, it's essential to test them to ensure that users and groups have the appropriate access to Jenkins projects.

**Creadetials to Integrate with other tools**

Navigate to**Jenkins Dashboard → Manage Jenkins → Security → Credentials → (global) → Add Credentials**

* + For "Username with password", you'll need to enter the username and password.
  + For "SSH username with private key", you'll need to provide the username and paste the private key.
  + For "Secret text", you'll enter the secret value.
  + For "Secret file", you'll upload the file containing the secret.

**Select Credential Type**: Choose the type of credentials you want to add from the dropdown menu. Jenkins supports various types of credentials, such as username with password, SSH username with private key, secret text, secret file, etc. Select the appropriate type for your use case.

**Save Credentials**: After entering the necessary details, click on the "OK" or "Save" button to add the credentials to Jenkins.

**Use Credentials in Jobs**: Once the credentials are saved, you can use them in your Jenkins jobs and plugins. During job configuration, you'll have the option to select the appropriate credential from the list of available credentials.

**Environment Variables**

In the Linux operating system, environment variables, or ENVs for short. The term "environment variable" refers to a dynamic-named value that can affect the way running processes will behave on a computer.

Use all uppercase letters for the environment variable name.

When specified in a terminal, a global environment variable is accessible from any location inside the terminal. This indicates that scripts, applications, or processes operating within that terminal's scope may utilize it.

Environment variables are used to store app secrets and configuration data

environment variables are commonly set and managed through shell scripts or configuration files like .bashrc, .bash\_profile, or .profile depending on the shell you are using

**To list all the environment variables in Linux:**  
printenv (OR) env

**To get an environment variable's value:**  
$VARIABLE

**To print an environment variable's value:**  
echo $VARIABLE

**To create an environment variable temporarily in Linux:**  
export VARIABLE="value"

**To Make Persistent Environment Variables (Permanent) in Linux**  
Add export VARIABLE="value" in ~/.bashrc

**To Delete Environment Variables in Linux**  
unset VARIABLE

**The most common environment variables include:**

**PWD** – the current working directory.

**USER**– the currently logged-in user account.

**SHELL** – the current user’s shell location.

**HOME** – the user’s home directory.

**EDITOR** – the system’s default file editor.

**PATH** – the location of all system executable files and It **specifies the directories to be searched to find a command.**

**tar**

tar is used for creating and extracting archive files.

tar command can create compressed or uncompressed archive files, as well as maintain and modify them.

**Syntax:**

tar [options] [archive-file] [file or directory to be archived]

     [options]

**-c**  Creates an archive

**-x** Extracts the archive

**-v** Displays verbose information

**-f**  Creates an archive with a specified filename

**-z**  Tells the tar command to create a tar file using gzip

More options:

**-t**Displays or lists files in an archived file

**-u** Archives and adds to an existing archive file

**-A** Concatenates the archive files

**-j**  Filters the archive tar file using tbzip

**-W** Verifies an archive file

**-r** Updates or adds a file or directory to an already-existing .tar file.

**example:**

**1.**  **Create a tar archive with gzip compression. (-c)**

tar -czvf <tar\_file\_name>.tar.gz <files or folder>

**ex:**tar -czvf jenkins\_backup.tar.gz /var/lib/jenkins

**2.**  **Extract a gzip compressed tar archive. (-x)**

tar -xzvf <tar\_file\_name>.tar.gz

**ex:** tar -xzvf jenkins\_backup.tar.gz

**zip**

The zip command is a popular tool for compiling and archiving files.

It is available on most operating systems, including Linux, macOS, and Windows.

The command allows you to create, view, and extract zip archives.

The basic syntax for creating a zip archive is as follows:

**1.  Create a zip archive.**

zip <zip\_file\_name>.zip <files or folder>

**2.  To extract the contents of a zip archive.**

unzip archive\_name.zip

A diagram of a software framework

Description automatically generated

A screenshot of a white background

Description automatically generated

**Sample python Flask application**

**STEP 1:  Install python3 and pip3**

sudo apt update -y

sudo apt install python3 python3-pip -y

pip3 install flask

**Note:**If we get "error: externally-managed-environment" -> Run below command

sudo rm /usr/lib/python3.12/EXTERNALLY-MANAGED

**STEP 2:  Create flask app in ec2 (vi app.py)**

from flask import Flask

import socket, json

app = Flask(\_\_name\_\_)

@app.route('/')

def print\_ip():

hostname = socket.gethostname()

get\_ip = socket.gethostbyname(hostname)

return get\_ip

@app.route('/health')

def health\_check():

return json.dumps({'success':True}), 200, {'ContentType':'application/json'}

@app.route('/name')

def print\_name():

return 'Harsha Jain'

@app.route('/error')

def print\_error():

print('500 error')

sum = 0/100

return sum

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host="0.0.0.0", port="8080")

**STEP 3:  Run the flask app.py**

python3 app.py

**STEP 4:  Access the flask app**

To access the flask app

Open 8080 port in ec2 security group

<ec2\_public\_ip\_address>:<port>/<path>

ex: 172.31.10.232:8080 - Request root of application which prints ip of ec2

      172.31.10.232:8080/name - Request /name path which prints 'Harsha Jain'

**Pre-requisits / Server setup**

**STEP 1: Update the ec2 and Install Java, Maven, Tomcat10**

# Update new Ubuntu ec2 machine

sudo apt udpate

# Install java development kit

sudo apt install -y openjdk-21-jdk

# Install maven

sudo apt install -y maven

# Install Apache Tomcat10

sudo apt install tomcat10 tomcat10-admin -y

#To check Tomcat running

sudo systemctl status tomcat10

**STEP 2: Add user to tomcat10 to access Manager GUI Pannel**

sudo vi /etc/tomcat10/tomcat-users.xml

# Add below line inside <tomcat-users> ..... </tomcat-users>

<role rolename="manager-gui"/>

<role rolename="admin-gui"/>

<user username="admin" password="password" roles="manager-gui,admin-gui"/>

**STEP 3: Create your own github repo with the java codebase which is in Applications/java\_app.zip** ([https://materials.itdefined.org](https://materials.itdefined.org/))

**STEP 4: Build the calculator.war which you will get inside target/calculator.war after running below maven command**

mvn clean package

Note: Run the above maven command in same directory where pom.xml is present

**STEP 5: Deploy the** **calculator.war manually for now through Manager GUI**

Access the Tomcat Manage GUI: <ec2\_public\_ip\_address>:8080/manager

User the username and password which we configure in STEP 2

Upload the war file



**STEP 6: Access the calculator app <ec2\_public\_ip\_address>:8080/calculator**

**Master - Slave as Permanent Agent**

**Configure Jenkins Slave:  
STEP\_1:**Create an ec2 instance (ubuntu ami)  and run the below commands.

**sudo apt update   
    sudo apt install -y openjdk-17-jdk**

**Configure Jenkins Master:  
STEP\_1:** Go to **"Manage Jenkins"** -> **"System Configuration"** -> **"Nodes"** -> "New Node".

a. Enter a node name

b. select "Permanent Agent" and Create

**STEP\_2:**Configure the following:

**a. Name:** "Name for the slave node (non - identifier)"

**b. Number of executors:** The maximum number of concurrent builds that Jenkins may perform on this node.

**c. Remote root directory:**An agent needs to have a directory dedicated to Jenkins. Specify the path to this directory on the agent. It is best to use an absolute path

**d. Labels:**Labels (or tags) are used to group multiple agents into one logical group. This is used as an identifier while selecting this node as an agent.

**e. Launch method:** "Launch agent via SSH"

**Host:** <ip\_address> of slave ec2 instance (private IP is better if the node master is in same network)

**Credentials:**Add new credentials

**kind:**  SSH Username with private key (ec2 -> username and private key: pem file content)

**Host Key Verification Strategy:** Non-verifying Verification strategy (Other we will learn after passwordless SSH)

**Note:** All other settings leave their default value

**f. Save**

**If Jenkins is getting very slow update the file /var/lib/jenkins/jenkins.model.JenkinsLocationConfiguration.xml with the current public\_ip of the ec2 instance**

<?xml version='1.1' encoding='UTF-8'?>

<jenkins.model.JenkinsLocationConfiguration>

<jenkinsUrl>http://<current\_ec2\_public\_ip>:8080/</jenkinsUrl>

</jenkins.model.JenkinsLocationConfiguration>

**To automate this public\_ip updation on every restart of ec2**

Schedule a bash script to update public\_ip

**STEP 1: Create a bash script (vi /home/ubuntu/update\_ip.sh)**

#!/bin/bash

echo "<?xml version='1.1' encoding='UTF-8'?>

<jenkins.model.JenkinsLocationConfiguration>

<jenkinsUrl>http://$(curl -s http://checkip.amazonaws.com):8080/</jenkinsUrl>

</jenkins.model.JenkinsLocationConfiguration>" > /var/lib/jenkins/jenkins.model.JenkinsLocationConfiguration.xml

**STEP 2: Schecdule above script on restart (crontab -e)**             @reboot sudo bash  /home/ubuntu/update\_ip.sh

**Jenkins Pipeline**  
   How to create a Jenkins pipeline job?  
    1. We can add pipeline configuration in 2 ways   
        a. We write all the configuration inside the pipeline job in job ui only.  
        b. We can write all the configuration in a file called Jenkinsfile and we can add that   
           from a SCM/Version control   
    2. We are using only the declarative pipeline type in our company.

Docs: <https://www.jenkins.io/doc/book/pipeline/syntax/>

pipeline {

agent any

stages {

stage('Checkout') {

steps {

echo "This is git clone stage"

sh "sleep 5"

sh "ls -lrt"

}

}

stage('Build') {

steps {

sh '''

echo Complie and build the package

sleep 5

'''

}

}

stage('Test') {

steps {

sh """

#!/bin/bash

echo This is Integration testing

sleep 5

ls -lrt

"""

}

}

}

}

**If Jenkins is getting very slow update the file /var/lib/jenkins/jenkins.model.JenkinsLocationConfiguration.xml with the current public\_ip of the ec2 instance**

<?xml version='1.1' encoding='UTF-8'?>

<jenkins.model.JenkinsLocationConfiguration>

<jenkinsUrl>http://<current\_ec2\_public\_ip>:8080/</jenkinsUrl>

</jenkins.model.JenkinsLocationConfiguration>

**To automate this public\_ip updation on every restart of ec2**

Schedule a bash script to update public\_ip

**STEP 1: Create a bash script (vi /home/ubuntu/update\_ip.sh)**

#!/bin/bash

echo "<?xml version='1.1' encoding='UTF-8'?>

<jenkins.model.JenkinsLocationConfiguration>

<jenkinsUrl>http://$(curl -s http://checkip.amazonaws.com):8080/</jenkinsUrl>

</jenkins.model.JenkinsLocationConfiguration>" > /var/lib/jenkins/jenkins.model.JenkinsLocationConfiguration.xml

**STEP 2: Schecdule above script on restart (crontab -e)**             @reboot sudo bash  /home/ubuntu/update\_ip.sh

**Jenkins Pipeline**  
   How to create a Jenkins pipeline job?  
    1. We can add pipeline configuration in 2 ways   
        a. We write all the configuration inside the pipeline job in job ui only.  
        b. We can write all the configuration in a file called Jenkinsfile and we can add that   
           from a SCM/Version control   
    2. We are using only the declarative pipeline type in our company.

Docs: <https://www.jenkins.io/doc/book/pipeline/syntax/>  
   What are the ways to create a Jenkins Job?    
   What is a Jenkinsfile?

**Jenkinsfile**      
    - It is the file in which we write all our job configurations in Groovy.  
    - In our company, we are using a declarative pipeline in Jenkinsfile and we are maintaining   
      Jenkinsfile through git.

**Types of pipeline jobs**  
    Jenkins pipeline uses Groovy scripting which   
    is one of the Java-based scripting languages.   
      
    2 types of pipeline job   
       **1.** **Scripted pipeline**  
        - Traditional way of defining pipeline Job   
        - Most of the configuration we need to define manually.   
        - All the code will be inside the node block.

node {

stage('Build') {

//

}

stage('Test') {

//

}

stage('Deploy') {

//

}

}

**2. Declarative Pipeline**     
        - New way of defining pipeline jobs.  
        - Most of the configurations are pre-defined.                                      
        - All the code will be inside the pipeline block.

pipeline {

agent any

stages {

stage('Build') {

steps {

//

}

}

stage('Test') {

steps {

//

}

}

stage('Deploy') {

steps {

//

}

}

}

}

**Agent**  
    - An agent is a declarative used to define the execution method for pipeline stages.  
    - agent is the one which tell stages to execute on which node and its executor.  
    - Agent can be specified at two levels   
                a. pipeline level   
                b. Stage level  
              
    **Types of agent**  
**1. any**  
            - This is the default agent, If I wont specify any agent this will be considered.  
            - Runs the pipeline stages on any available node executor.

pipeline {

agent any

stages {

stage('Build') {

steps {

sh 'echo "Building..."'

}

}

stage('Test') {

steps {

sh 'echo "Testing..."'

}

}

stage('Deploy') {

steps {

sh 'echo "Deploying..."'

}

}

}

}

**2. label**  
            - we can define agent as label at pipeline to run all the stages of pipeline on a particular node with label.  
            - For agent at stage level use agent none

# Agent Label at Pipeline level

pipeline {

agent {

label 'master'

}

stages {

stage('Build') {

steps {

sh 'echo "Building..."'

}

}

}

}

**3. none**   
            -  To define executor for each and every stage declare agent as none at pipeline level.  
            -  Using this we specify Jenkins that agent is controlled at each stage level.  
            -  Defining agent at stage level is mandatory. 

pipeline {

agent none

stages {

stage('Build') {

agent {

label 'docker' // Specify a specific agent label for this stage

}

steps {

sh 'docker build -t myapp .'

}

}

stage('Test') {

agent {

label 'docker' // Specify the same agent label for consistency

}

steps {

sh 'docker run myapp test'

}

}

stage('Deploy') {

when {

branch 'master'

}

steps {

sh 'docker push myapp:latest'

}

}

}

}

**GitHub Source:**[**https://github.com/jaintpharsha/jenkins\_pipeline\_jobs\_may24.git**](https://github.com/jaintpharsha/jenkins_pipeline_jobs_may24.git)

**git / checkout**

The **git** step is used to specify the Git repository from which you want to fetch the source code.

The **checkout** step is used to check out code from source control (not limited to Git repositories) into the workspace of the agent. It’s typically used when you need more control over the checkout process, such as checking out specific branches or commits.

pipeline {

agent any

stages {

stage('Checkout') {

steps {

git branch: '<branch\_name>', credentialsId: '<cred\_id>', url: '<git\_repo\_url>'

}

}

// other stages can follow...

}

}

pipeline {

agent any

stages {

stage('Checkout') {

steps {

checkout([$class: 'GitSCM',

branches: [[name: '\*/master']],

doGenerateSubmoduleConfigurations: false,

extensions: [],

submoduleCfg: [],

userRemoteConfigs: [[url: 'https://github.com/username/repo.git']]])

}

}

// other stages can follow...

}

}

**environment**

The environment directive specifies a sequence of key-value pairs which will be defined as environment variables for all steps, or stage-specific steps, depending on where the environment directive is located within the Pipeline.

pipeline {

agent any

environment {

APP = 'frontend'

DB\_URL = '192.168.28.24'

BRANCH = 'main'

GIT\_URL = 'https://github.com/harshaprakash100/ip\_app.git'

CRED\_ID = 'github\_hp'

}

stages {

stage('Environment Variable') {

steps {

script {

echo "${env.APP}: ${env.DB\_URL}"

}

sh "echo ${APP}"

}

}

stage('checkout') {

steps {

git branch: "${env.BRANCH}",

credentialsId: "${env.CRED\_ID}",

url: "${env.GIT\_URL}"

}

}

stage('check checkout') {

steps {

sh 'ls -lrt'

}

}

}

}

**when**

The when directive allows the Pipeline to determine whether the stage should be executed depending on the given condition. The when directive must contain at least one condition.

**1. environment** Execute the stage when the specified environment variable is equal to the given value

when {

environment name: 'DEPLOY\_TO', value: 'production'

}

**2. expression** Execute the stage when the specified Groovy expression evaluates to true

//Execute the stage when the specified Groovy expression evaluates to true

when {

expression {

return params.DEBUG\_BUILD

}

}

//Execute the stage when the parameter DEBUG\_BUILD is true

when {

expression {

params.DEBUG\_BUILD == true

}

}

**3. branch**Execute stage only if the current branch matches.  
    **not**Execute the stage when the nested condition is false.  
    **anyOf** Execute the stage when at least one of the nested conditions is true.

// Execute stage when branch is 'master'

when {

branch 'master'

}

// Execute stage when branch is not 'master'

when {

not {

branch 'master'

}

}

//Execute when matches all the branches

when {

anyOf {

branch 'master'; branch 'staging'

}

}

**5. equals** Execute the stage when the expected value is equal to the actual value

when {

equals expected: 2,

actual: currentBuild.number

}

**6. allOf** Execute the stage when all of the nested conditions are true

**parameters**

The parameters directive provides a list of parameters that a user should provide when triggering the Pipeline.

Type of parameters   
**String Parameter:**This is a basic parameter that allows users to input a single line of text. It can be useful for providing simple inputs such as file paths, version numbers, or any other single line of text.

**Boolean Parameter**: This parameter type provides a checkbox in the Jenkins build interface. It's used when you want to provide a simple true/false or yes/no option to the user.

**Choice Parameter**: This parameter type presents a dropdown menu to the user, allowing them to select one option from a predefined list of choices. It's useful when you have a specific set of options to choose from.

parameters {

string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should I say hello to?')

text(name: 'BIOGRAPHY', defaultValue: '', description: 'Enter some information about the person')

booleanParam(name: 'TOGGLE', defaultValue: true, description: 'Toggle this value')

choice(name: 'CHOICE', choices: ['One', 'Two', 'Three'], description: 'Pick something')

password(name: 'PASSWORD', defaultValue: 'SECRET', description: 'Enter a password')

}

**Options**

Options directive is used to define various pipeline-wide options.

pipeline {

agent any

options {

// Define options here

}

stages {

// Define stages here

}

}

**Timeouts**: You can define timeouts for various stages or the entire pipeline.

options {

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

}

**Build Discard:**Specifies how Jenkins should discard old builds to conserve disk space.

options {

buildDiscarder(logRotator(numToKeepStr: '10'))

}

**Disable Concurrency**: Ensures that only one instance of the pipeline can run at a time.

options {

disableConcurrentBuilds()

}

**retry:** On failure, retry the entire Pipeline the specified number of times.

options {

retry(4) // Retry 4 times if job fails

}

**triggers  
The triggers directive defines the automated ways in which the Pipeline should be re-triggered.**

**cron:** Accepts a cron-style string to define a regular interval at which the Pipeline should be re-triggered.

triggers {

cron('H \* \* \* \*')

}

**pollSCM:** Accepts a cron-style string to define a regular interval at which Jenkins should check for new changes in git.

triggers {

pollSCM('H \*/4 \* \* 1-5')

}

**Pipeline or Job Dependencies**: This triggers the current pipeline job based on the successful completion of another job.

triggers {

upstream(upstreamProjects: '<other\_job\_name>', threshold: hudson.model.Result.SUCCESS)

}

**catchError**

Catch errors and set build result and stage result to SUCCESS, FAILURE, UNSTABLE

stage('Test1') {

steps {

catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE') {

sh '''

sleep 10

exit 1

'''

}

}

}

**Parallel**

Stages in Declarative Pipeline may have a parallel section containing a list of nested stages which will be executed in parallel.

stage('Parallel Stage') {

parallel {

stage('A') {

steps {

echo "On Branch A"

}

}

stage('B') {

steps {

echo "On Branch B"

}

}

}

}

**script**

The script step takes a block of the Scripted pipeline and executes that in the Declarative Pipeline.

If we want to use the if condition, for and while loop in the Declarative pipeline, we need to call it inside the script block.

stage('NAME') {

steps {

script {

if ( params.DEPLOY == 'true') {

.......... <some\_login>

}

}

}

**Error handling**

error handling is typically achieved using the try and catch blocks, which provide a structured way to manage errors and exceptions that may occur during the execution of pipeline stages or steps.

pipeline {

agent any

stages {

stage('Example') {

steps {

script {

try {

// Code that might throw an exception

sh 'some-command-that-might-fail'

} catch (Exception e) {

// Handling the exception

echo "Caught an exception: ${e.message}"

// You might also want to mark the build as failed

currentBuild.result = 'FAILURE'

} finally {

// Optional: Code that will be executed regardless of whether there was an exception or not

echo "Cleaning up resources..."

}

}

}

}

}

}

**Post Actions**

post actions are used to define actions that should be performed after the completion of all stages in the pipeline, regardless of whether the pipeline succeeds or fails. Post actions allow you to define cleanup tasks, notifications, or any other actions that should occur once the pipeline execution finishes.

pipeline {

agent any

stages {

// All the required stages

............

}

post {

always {

// Actions to be performed always (whether the pipeline succeeds or fails)

echo 'Pipeline finished. Performing cleanup tasks...'

cleanWs() // Example: Cleanup workspace

// Send notifications, archive artifacts, etc.

}

success {

// Actions to be performed if the pipeline succeeds

echo 'Pipeline succeeded! Sending success notifications...'

}

failure {

// Actions to be performed if the pipeline fails

echo 'Pipeline failed! Sending failure notifications...'

}

unstable {

// Actions to be performed if the pipeline is unstable

echo 'Pipeline is unstable! Taking necessary actions...'

}

changed {

// Actions to be performed if the pipeline status changed

echo 'Pipeline status changed! Executing appropriate actions...'

}

}

}

**A close-up of a sign

Description automatically generated**



The Stage retrieves latest code from Git repo by using GitSCM plugin, which clones the repository from url provided with default branch or we can checkout required branch.

A green rectangle with black text

Description automatically generated

Execute unit tests to verify the functionality of individual components or modules within the application.

Unit testing is performed using Maven's **mvn test** command, which runs tests located in the src/test/java directory.

A green rectangle with black text

Description automatically generated

This stage verifies the interaction between different components or services within the application.

Integration testing is executed with Maven's **mvn integration-test** command.

It validates how various modules or services work together after integrating the changes from multiple developers to achieve the desired functionality.

**A green rectangular object with black text

Description automatically generated**

Conduct performance tests to evaluate how the application handles a specific workload or user load.

Load testing is performed using Maven's **mvn clean verify** command.

It simulates multiple users accessing the application concurrently to assess its response times and scalability using JMeter.

**A screenshot of a computer

Description automatically generated**

Deploy the application to a Tomcat 10 server instance for hosting and accessibility.

Deployment is managed through Jenkins using a Tomcat manager adapter.

A green rectangle with black text

Description automatically generated

**Clean Workspace (cleanWs()):** After completing the pipeline stages, this action cleans up the Jenkins workspace, removing temporary files and ensuring a clean environment for future builds.

**Sending Notifications:** Notifications can be sent to team members or stakeholders via email, Slack, or other messaging platforms to inform them about build results or status.

**Publishing Reports:** Jenkins can publish various reports generated during the build, such as test results, code coverage reports, and static code analysis reports.

**Triggering Downstream Jobs:** If your pipeline is part of a larger workflow, successful builds can trigger downstream jobs or pipelines for further integration or deployment. For pipelines handling deployment, a promotion step can be added to promote the artifact to production after passing all necessary tests and validations.

**ASSIGNMENT**

**Configure Discard Old build:** Keep the most recent 5 builds and automatically delete all older builds.

**Setup PollSCM for Java App git repo:** When a code change occurs (new commit), the pipeline job should trigger automatically, and the updates should be visible in the Tomcat or web application environment.