JENKINS

Contents

JENKINS

1) Jenkins

- Jenkins is an open-source automation server widely used for implementing continuous integration (CI) and continuous delivery (CD) pipelines.
- It automates tasks throughout the software development lifecycle, streamlining(quicker) development and deployment processes.
  - CI/CD → automated process in software development. It involves building, testing, and deploying code.

[Note:
★ Jenkins Homepage is also known as **Jenkins dashboard**.
★ In Jenkins, J0B == PROJECT == ITEM → All Are same.
★ In Jenkins, running a J0B is called **BUILD**.
★ while running job
  - if job succeeded → build is success
  - if job failed → build is failed.

★ Jenkins default home directory path → **/var/lib/Jenkins**
★ Jenkins default port number is **8080**
  ★ Jenkins Login Page → instanceIP:8080
★ https://updates.jenkins-ci.org/download/plugins/ → website for plugins
★ https://www.jenkins.io/doc/book/pipeline/syntax/ → pipeline documentation

2) Master & Slave in Jenkins

Describe: Jenkins utilizes a master-slave architecture.

- **Master:** The central server manages jobs, assigns them to slaves, and provides the web interface for user interaction.
- **Slave (Agent/Node):** execute the actual build and follows instructions given by master.

Why do we need to use master/slave setup?

To reduce workloads on master server and utilizing multiple slave nodes, Jenkins can handle concurrent builds, increasing capacity and speeding up job execution.

3) Jenkins Setup

(a) To use Jenkins, create instance in EC2 in AWS console,

For instance creation choose t2.large, (t2.micro → free tier)

[Note: Since, t2.large is chargeable so we can choose t2.micro but if we want all packages to run then we need to choose t2.large]

Once instance is created then connect to that instance in ubuntu.

(b) Install Jenkins:

Step1	→	Login to ubuntu user in Gitbash
Step2	→	https://www.jenkins.io/doc/ (website to download java and Jenkins)
Step3	→	go to "install Jenkins" → Linux → copy paste in Gitbash (a) install java & (b) install Jenkins in order as shown below **(a) Install Java** sudo apt update sudo apt install fontconfig openjdk-17-jre java -version openjdk version "17.0.8" 2023-07-18 OpenJDK Runtime Environment (build 17.0.8+7-Debian-1deb12u1) OpenJDK 64-Bit Server VM (build 17.0.8+7-Debian-1deb12u1, mixed mode, sharing) **(b) Install Jenkins (Long Term Support Release)** 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 jenkins
Step4	→	check status: Syntax: sudo systemctl status jenkins.service (or) sudo service Jenkins status

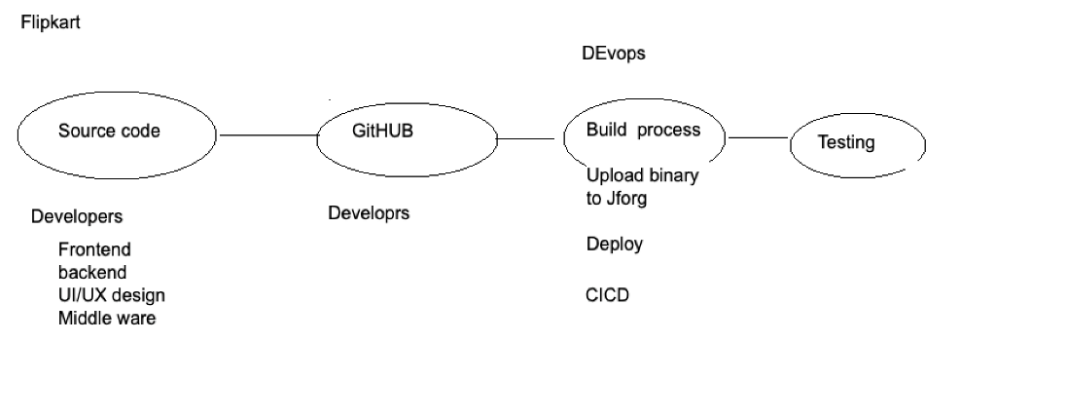
(c) To open Jenkins service tab,

- copy paste instance private IP in url of browser and give **default port number :8080** after IP (i.e, IP:8080)
- To open the server, we need to **add inbound rule:**
  Tick instance → security → click on security groups → edit inbound rules → Add rule → Type (All traffic), source(anywhere-IPv4) → save
- Now, go to Jenkins tab → refresh page (we can see Jenkins server page)

(d) Find password to authenticate Jenkins service,

- Since, it's password protected we need to give password:
- Copy "path" for password and open Linux terminal (Gitbash)
  Path → (/var/li/jenkins/secrets/initialAdminnPassword)
- syntax: sudo cat <path> → displays password
  (sudo cat /var/li/jenkins/secrets/initialAdminnPassword)
- Copy paste password in Jenkins tab → Jenkins will be opened
- Create user, password, full name and mail ID → log into Jenkins

➢ **Build Process:** source code to file format, .file extension like .exe, .dmg (WhatsApp format), etc.



4) *Jenkins configurations*

Jenkins Dashboard → Manage Jenkins

(a) System configuration

- **System** → integrate tools with Jenkins
- **Tools** → multiple versions in build in tools
- **Plugins** → add /remove plugins as per requirements
- **Nodes** → distribute the load (share Jenkins load)
- **Clouds** → To Integrate Kubernetes clusters that acts as agents/nodes. Instead of nodes Kubernetes will create parts so that parts will be acting as agents.

(b) Security:

- login related and permissions things
- **Security:** Can add users and groups so we can give permissions of access for that user/group
- **Security realm:**
  - **LDAP:** server where group of users will be available in organization
  - **Jenkins own user database:** for practise purpose.

(c) Authorization:

- **Matrix based security:** to give permissions for users to access
- Once, LDAP is selected then groups like Developers, testers. Etc will be displayed after which we can give any permission as per requirement.
- Since, for practice we can check with creating users and giving permissions.
  - eg) if we give all permissions to user/group then they will be able to do any operations in Jenkins.
  - If we restrict and give few permissions like read then they will be able to just read, can't execute or modify anything.
  - Security used for giving permissions to groups/users.

(d) Create project:

Dashboard → New item → Enter an item name (select folder) → ok

(e) To enable properties (Enable project-based security) in project config:

Dashboard → Manage Jenkins → security → Authorization (Project-based matrix authorization strategy → add user (give required permissions) → save

(f) Giving security(permissions) for project:

Dashboard → Project A → Configuration → Enable project-based security → add user (give required permissions) → save

(g) Add credentials:

Dashboard → Manage Jenkins → Credentials → System → Global credentials (unrestricted) → add credentials

(h) Nodes/Agents: (Master slave concept)

Used to distribute the load. Agents are known as slaves.

Create nodes/agents: [one node - one instance in AWS]

Dashboard → Nodes → New Node → Node Name (any name), select permanent agent → create → name (name of agent/node), no of executers (No of jobs at a time like 1,2,etc), Remote root directory (/home/ubuntu), Launch method (select "launch agents via ssh" → Host (instanceIP), credentials (add → Id (Id name like ssh_key_jenkins), user(ubuntu), Private key (select Enter directly → copy paste key of pem file (cat pemfile.pem in local user in Gitbash) → add), Host key verification strategy (Non verifying verification strategy) → save

[**Note:** if we give no of executers as 2 in agent then agent can run 2 jobs at a time, if we give crontab to the job then that job will run in 1executor only]

(i) Labels:

- Labels are attributes or tags assigned to both agents (nodes) and jobs in Jenkins.
- used to restrict any agents or jobs so those jobs will run in that agent only. If we give 2 jobs for same agent then both jobs will run in same agent, until 1st job is completed executing, 2nd job will be waiting.
- If we don't give any labels while creating nodes then jobs will run in any agents which is free, load is distributed among agents.
- So, in organization level jobs will be distributed to agents by giving labels (divide the agents with labels). eg) Java job for 1 agent, Python job for another agent, etc… (dedicated job for dedicated agents)

(j) Create Jobs:

Jobs are user-defined configurations specifying a set of tasks to be executed in a specific order. Can create jobs anywhere, either inside folder or in dashboard.
[Dashboard → New item → Enter an item name (Job name) → type of job → ok]

Types:

i) **Freestyle Project:** The most common type, offering flexibility to define build steps using shell commands or plugins. (Great for beginners)
ii) **Pipeline:** Jobs written in a Groovy-based domain-specific language (DSL) for more structured and declarative pipelines. (Ideal for complex workflows)
iii) **Maven Project:** Specifically designed for building and testing Maven projects.

[**Note:** freestyle projects are old & traditional method, have less options and functions whereas pipeline have more features and integrates CICD pipeline]

(k) Configure job:

Dashboard → job_name → configure → build environment → Build steps (Add → execute shell → write codes) → ok

(l) Run the job:

Dashboard → job_name → build now → job will be executed → click on executed job (build up) to get into console output

5) ***Check workspace***

(i) Linux

- /var/lib/Jenkins → default path of Jenkins
- Ubuntu user → cd /var/lib/Jenkins (goes to Jenkins path)
- ls → lists everything related to Jenkins like files, directories, agents/nodes, Plugins, users, jobs, etc.
- cd data_name → To check any of these data information, go inside and list (ls)

(ii) Jenkins

Dashboard → job_name → workspace (can see directories/files created in the job)

6) Configuration with respect to job

Freestyle job is old traditional job method which have limited options instead can use pipeline jobs which supports lot of plugins which is used to automate and configure.

(i) General tab:

(a) Enable project-based security: restrict the users and give permissions for users

(b) Discard old builds: Deletes old builds, we can give after how many days our old builds should get deleted and also can select how many builds can be available.

[Note: In organization level, usually 25-30 builds are kept and for 30 days so after 30days all the builds will be deleted.]

(c) GitHub projects: provide GitHub repository project URL, used while configuring and using pollSCM

(d) The project is parameterised: provide parameters like string, Boolean, choice, password, etc as per requirements.

Used to give parameter while running scripts so based on inputs, stages will run.

(e) Throttle builds: skip unnecessary builds, if one build is running and if others build have started then it will ignore. We can set number of builds and time so can restrict more builds within given time.

(f) Execute concurrent builds if necessary: similar to throttle builds

[Note: Throttle builds & execute concurrent builds are used for controlling builds]

(ii) Build Triggers:

(a) Trigger builds remotely: if we enable tigger then we need to give token (any random token), while executing job then tokens should match

(b) Build after other projects are built: by giving job_name here, once any other jobs are running then as soon as that job gets executed then this job (job_name) will get started automatically.

(c) Build periodically: automatically triggers job on scheduled time (crone jobs)

(d) Poll SCM: similar to build periodically but it triggers with respect to GitHub (need to configure GitHub), if any commits happen within linked repository in GitHub, then it auto triggers that job within that scheduled time. If we schedule job to run for every 30 min, then it keeps on checking every 30min, any commits happen within that time then job will be auto triggered.

[Note: we execute Build periodically when we want to give schedule day time, we execute Poll SCM when we want to execute any commits]

(iii) Build Steps:

(a) Delete workspace: deletes workspace (Folder, deletes all data)

(b) Secret texr (or) files: can add secret of files like SSH private key, secret text, secret file, username and password, etc

(c) Add timestamps to console o/p: adds timestamp in the console (at what time jobs are running)

7) Upstream and downstream jobs

Upstream jobs are those that trigger other jobs upon completion. Downstream jobs are triggered by upstream jobs.

This setup allows for the creation of job chains where the completion of one job can trigger subsequent jobs, facilitating (making easy) complex workflows.

8) Pipeline

Pipeline is sequence of activities that Jenkins run to achieve specific goal such as build, test and deploy an application, represents the CI/CD process as code.

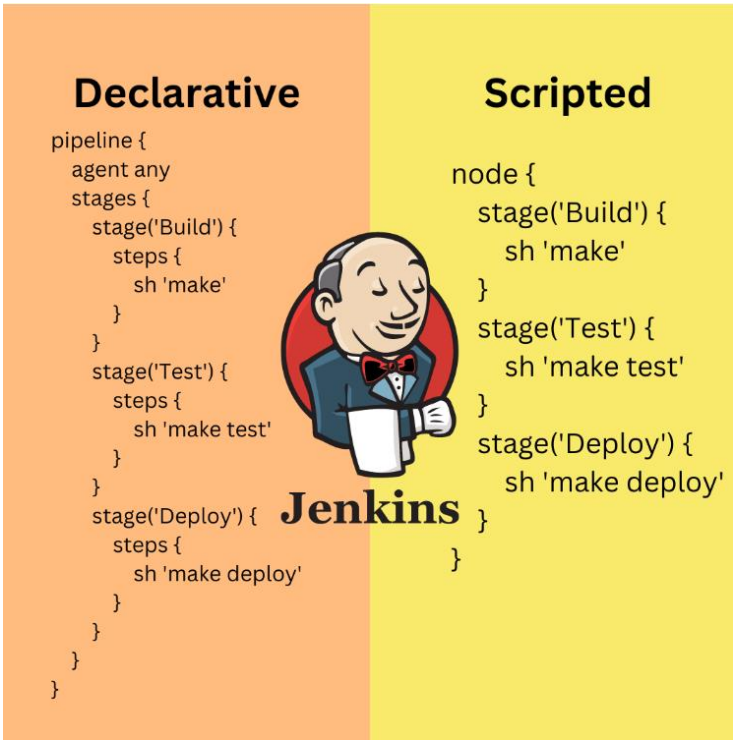
- The structure typically includes:
  - **Stages:** divisions within pipeline and defining logic phases of entire pipeline
  - **Steps:** Individual tasks within a stage

- ***[Create pipeline:***
  Dashboard → New Item → Enter an Item name (job name) → choose Pipeline → ok]
- ***Advantages:*** automate tasks**,** Divide the jobs like if we have different agents then if in scripts, we create many stages then those stages will be divided into jobs and execute parallelly in other agents that's available.

  Types

  a) Scripted pipeline: old method, everything will be declared under stages. Its complex to learn and remember.

  b) Declarative pipeline: mostly used and preferred by everyone.
  Structure: starts with pipeline, agents, stages, steps, then script/code. Can create multiple stages which acts like jobs.



(i) Agents

Agents (or nodes) referring to the slave machines that execute jobs. They are used to distribute the build load. one job can have multiple agents and jobs can be divided(stages) and run parallelly in other agents.

(ii) Stages

divisions within pipeline and defining logic phases of entire pipeline.
If we give "agent name" then that stage executes in that agent, if we give as "agent any" then it randomly selects agents which is free.

(iii) Steps

Individual tasks within a stage. When a step succeeds it moves onto the next step. When a step fails to execute correctly the Pipeline will fail.

- ***Comment*:** used to comment out a block of code
  [**Note:** // → single line comments out
  /* …….. */ → multiple line comments out]

(iv) Reply

Allow users to make temporary changes in a pipeline script without affecting original pipeline.
Benefits: Useful for troubleshooting failed builds, reproducing successful builds for verification, or testing different configurations.

- ***Pipeline overview:*** can see all stages here and it's called blueprint, we can pause, restart from particular stage.
  [Note: We can either give specific agents "agent_name" for the stage (or) if we want stage to select any agent that's available then we need to give as, "agent any".
  - Under the stage, we define the steps what we want to achieve.
  - Under the steps, there will be process(code) that we want to achieve like creating file, directory, displaying, listing, etc.
  - Can use either echo or print to print the statement in the console.
  - cleanWs(): (cleanWorkspace)
    function used to clean workspace and start that stage from beginning]

(v) Labels

- Labels are attributes or tags assigned to both agents (nodes) and jobs in Jenkins.
- If we don't give any labels while creating nodes then jobs will run in any agents which is free, load is distributed among agents.

(vi) Plugin

- **Purpose:** Extend Jenkins' functionality with features for specific tools, technologies, and integrations.
- **Examples:** Git plugin for Git integration, Maven plugin for Maven support, AWS plugin for Amazon Web Services interactions.

9) Diff between 2 types of pipelines

i) **Declarative Pipeline:** Uses a more structured and simpler syntax designed for readability and maintainability. Declarative Pipeline is designed to be more readable and easier

ii) **Scripted Pipeline:** These are the old and traditional way of job configuration we have limited options here compared to declarative pipeline. The scripted pipeline will start with node. Scripted Pipeline can be more complex.

iii) Readability: Declarative Pipeline is designed to be more readable and easier to understand, while Scripted Pipeline can be more complex

iv) Syntax: The syntax for Scripted Pipeline is based on Groovy scripting language, while the syntax for Declarative Pipeline is also based on Groovy, but it uses a more structured and predefined format.

v) Flexibility: Scripted Pipeline provides more flexibility and control over the pipeline workflow, while Declarative Pipeline provides a simpler and more structured syntax.

10) Variables

(a) User defined (*environment variables*): Custom variables that we define in the pipeline to store values that can be reused across the pipeline.

```
eg) environment {
        tool_name : 'jenkins'
        }
```

(b) Pre-defined: Built-in environment variables provided by Jenkins, such as BUILD_NUMBER, JOB_NAME, and BRANCH_NAME.

[*declaring instanceIP:8080/env-vars.html/* → to check pre-defined variables which can be used in scripts.]

eg) JOB_NAME → shows job name

BUILD_NUMBER → shows build no

[Note: https://www.jenkins.io/doc/book/pipeline/syntax/ → pipeline documentation]

eg) User defined and pre-defined

```
pipeline {
   agent any
   environment {
      tool_name = 'git'
          }
   stage('test1') {
      steps {
         print tool_name                    #user defined var
         sh "echo ${tool_name}"
         print BUILD_NUMBER                 #pre-defined var
         sh "echo ${BUILD_NUMBER}"
         print JOB_NAME
```

```
            sh "echo ${JOB_NAME}"
            print NODE_NAME
            sh "echo ${NODE_NAME}"
        }
    }
  }
}
```

11) Agent

a) **any:** available agent will automatically select
   Syntax: agent any

b) **none:** will be used to schedule the stage on different agents using label on each stage
   Syntax: agent none

c) **label:** schedule on particular agent
   Syntax: agent {label 'agent_name'}

d) **docker:** used to run on docker containers

[Note1.1: If we want to use multiple agents then best option is agent none]

[Note1.2: we should declare agents inside stage using label.
If we declare agents using label before stage (i.e, at starting) then all stages will be executed in that agent, but we can also declare different agent inside stage which will be executed in that agent for that stage]

eg1) declaring agents globally (before stage): all stages will run in specified agent

```
pipeline {
   agent { label 'agent1' }       #declaring agents globally
   stages {
      stage('Hello') {
         steps {
            cleanWs()
            sh "mkdir test"
            sh "echo 'hello world' > test/file1"
            sh "cat test/file1"
         }
      }
      stage('test') {
         steps {
            // print toll_name
            print BUILD_NUMBER
            print JOB_NAME
```

```
            print NODE_NAME
          }
        }
      }
    }
```

eg2) <u>declaring agents globally (before stage) and also locally (inside stage):</u>

all stages will run in the agent that's declared globally but if it's declared locally then that stage will run in the agent that's declared locally.

```
pipeline {
  agent { label 'agent1' }                  #declaring agents globally
  /*environment {                           #multiple line comment out
    toll_name = 'git'
  }*/
  stages {
    stage('Hello') {
      steps {
        cleanWs()
        sh "mkdir test"
        sh "echo 'hello world' > test/file1"
        sh "cat test/file1"
      }
    }
    stage('test') {
      agent { label 'agent2' }              #declaring agents locally
      steps {
        // print toll_name
        print BUILD_NUMBER
        print JOB_NAME
        print NODE_NAME
      }
    }
  }
}
```

eg3) <u>declaring agents locally (inside stage):</u>

if we want to declare all agents locally then we should declare global agent as " agent none" after which we can declare agents locally for all stages individually.

```
pipeline {
   agent none              #declaring agent none so can schedule the stage on
    different agents
   stages {
      stage('Hello') {
         agent { label 'agent1' }              #declaring agent locally
         steps {
            cleanWs()
            sh "mkdir test"
            sh "echo 'hello world' > test/file1"
            sh "cat test/file1"
         }
      }
      stage('test'){
         agent { label 'agent2' }              #declaring agent locally
         steps {
            print BUILD_NUMBER
            print JOB_NAME
            print NODE_NAME
         }
      }
   }
}
```

12) ***Parameters***

Define dynamic values that can be passed to a job or pipeline at runtime, allowing for customization without changing the core configuration.

```
Syntax: Parameters {
          # Declare parameters
          }
```

- When we declare parameters then build now becomes build with parameters, once we run that then it will ask for parameter to pass (like Jenkins, git, etc,)
- When we declare any parameters then configuration of parameters (This project is parameterized) will be automatically configured.
- We need to pass parameters globally, before stages.

[Note: difference between environment var (user defined var) and parameters is once environment var is defined globally then it executes the same but if we are defined then parameter will be asked to pass at runtime after which it will execute]

Types of parameters:

(i) String: used to pass string like characters

Syntax:

Declare: string (name: 'tool_name', defaultValue: 'jenkins', description: '')

Print: print tool_name

(ii) Text: used to pass any texts like one, two, etc…

Syntax:

Declare: text (name: 'DEPLOY_TEXT', defaultValue: 'One\nTwo\nThree\n', description: '')

Print: print DEPLOY_TEXT

(iii) Boolean: used to check Boolean value like True or false

Syntax:

Declare: booleanPar (name: 'DEBUG_BUILD', defaultValue: true, description: '')

Print: print DEBUG_BUILD

(iv) Choice: used to choose any choice that can be given in default value and choose which is in dropdown in runtime, that choice will be executed

Syntax:

Declare: choice (name: 'CHOICES', choices: ['one', 'two', 'three'], description: '')

Print: print CHOICES

(v) Password: used to set password

Syntax:

Declare: password (name: 'PASSWORD', defaultValue: 'SECRET', description: 'A secret password')

Print: print PASSWORD

(vi) file: used to import file from local system to Jenkins workspace

Syntax: file (name: 'file', description: '')

eg) Parameters: Declaring different Parameters and printing it

```
pipeline {
    agent any
    parameters{                          # declaring parameters
        string(name: 'toll_name', defaultValue: 'jenkins', description: '')
        text(name: 'DEPLOY_TEXT', defaultValue: 'One\nTwo\nThree\n',
     description: '')
        booleanParam(name: 'DEBUG_BUILD', defaultValue: true, description: '')
        choice(name: 'CHOICES', choices: ['one', 'two', 'three'], description: '')
```

```
    password(name: 'PASSWORD', defaultValue: 'SECRET', description: 'A
 secret password')
    file(name: 'file', description: '')
  }
  stages {
    stage('test') {
      steps {
        print toll_name              #print types of parameters
        print DEPLOY_TEXT
        print DEBUG_BUILD
        print CHOICES
        print PASSWORD
        print file
      }
    }
  }
}
```

13) Options

used to configure the Pipeline-specific options and change some pipeline behaviours.

Syntax: options {
#Declare option
}

Types:

(i) retry:

Retries the entire pipeline a specified number of times if it fails

Syntax: options {
retry (3)
}

[Note: In organizational level Retry can be used when any network issue is there, if we want to run/start any services, download anything, transfer huge files, clone to jenkins]

(ii) buildDiscarder:

Used to keep no of builds and discard old builds

(discard old builds in jenkins job config)

Syntax: options {
buildDiscarder(logRotator(numToKeepStr: '1')) #keeps 1 build
}

(iii) disableConcurrentBuilds:

Prevents concurrent builds of the pipeline. If there are 10 builds then 1 build will be executing, remaining builds will become one build and get executed.

```
Syntax: options {
        disableConcurrentBuilds()
        }
#queue all builds into one build until exiting build is executed
```

(iv) timeout:

Sets a maximum time for the pipeline to run. If the pipeline exceeds time, it will abort.

```
Syntax: options {
        timeout (time: 1, unit: 'HOURS')        #wait for 1hour
        timeout (time: 1, unit: 'MINUTES')      #wait for 1minute
        }
```

(v) timestamps:

Add current time in the console output.

```
Syntax: options {
        timestamps ()
        }
```

eg) Options:

giving retry option so job will retry(n) "n" no of times where n = 1,2, 3,

```
pipeline {
  agent any
    options {                                  # declaring option (retry) in stage level
      retry (3)                                                    # retry 3 times
        buildDiscarder(logRotator(numToKeepStr: '10'))          # keep latest 10
   builds
    disableConcurrentBuilds()   #1st build run,remaining n builds merged into
   1 build
    timeout(time: 1, unit: 'MINUTES')     # wait for 1min, if not executed will
   aborted
    timestamps()                           # displays current time of execution
    }
  stages {
    stage('test'){
      steps{
        retry (3) {                      #declaring option (retry) in step/job level
        print tool
        print toll_name
        print DEPLOY_TEXT
        print DEBUG_BUILD
```

```
                print CHOICES
                print PASSWORD
                print BUILD_NUMBER
                print JOB_NAME
                print NODE_NAME
                retry (3) {                    #declaring option (retry) for single line
                sh "mkdir test"
                }
               }
              }
             }
            }
           }
```

14) Validate or find syntax for any operation

Dashboard --> <pipeline_job_name> --> Pipeline Syntax
eg) Git
Sample step (Git) --> repository URL (GitHub account HTTPS URL) --> branch (<branch_name>) --> credentials [add --> username (GitHub username) --> password (GitHub settings --> developer settings --> personal access token --> tokens (classic) --> generate new token --> Note (any name), expiration (no exp), tick all --> generate token) --> copy paste token in password in Jenkins credentials --> ID (any ID like githubtoken) --> add] --> select github token from dropdown --> generate pipeline script (syntax/script generated) --> copy paste syntax in script --> build
Build the job: once job is executed then all files in GitHub will be in job workspace

15) Trigger another job within Jenkins

- We can trigger another job in the pipeline job by specifying the job name that we want to trigger.
- For example, we can link job1 to job_pipeline. Once we run job_pipeline, job1 will be triggered automatically, and a build will be created in job1.
- This is within the jenkins, outside Jenkins have different steps to be followed.

Syntax: build (<'job_name'>)
eg) build ('job2')

eg) Connecting to GitHub and triggering job within Jenkins

```
pipeline {
  agent any
  stages {
    stage('Hello') {
      steps {
        build ('job2')
      }
    }
  }
}
```

16) Schedule

- Minute (0-59) – what minute of the hour the job will run on.
- Hour (0-23, where 0 is midnight) → what hour the command will run in the 24-hour.
- Day of the month (1-31) – the day of the month we want the job to run on (g., the first of each month).
- Month (1-12) – tells the month when the job should run. We can number or name the month as well. (i.e., April, May, etc., could also be written as 4, 5, 6, etc.).
- Day of the week (0-7, where 0 or 7 are both Sunday) – the day of the week when we want to run our job. It can also be a number or name.

eg) */30 * * * * --> build every 30 minutes
0 */6 * * * --> build every 6 hours
15 * * 6,7 --> build every Saturday and Sunday at 3:00 pm

a) Cron:

- used to schedule job to run automatically for every minute, hours, days, month and day of the week respectively.
- Utilizing CRON jobs within Jenkins is a great way to automate the jobs.
- Used for performing regular scheduled actions such as backups, report generation, and so on.

Syntax: triggers { cron ('* * * * *') }

b) pollSCM:

- Used to schedule job to run automatically when any commits happen within specified minutes, hours, days, month and day of the week respectively. If no commits happen then job won't get triggered or run.
- It's used for GitHub in general, when any commits happen in GitHub within specified time then job gets automatically triggered and will build.
- Need to configure GitHub Project to link GitHub repository.

Syntax: triggers { pollSCM ('* * * * *') }

[Note: Not used much in organizational level, either build periodically (cron job) or git hooks are majorly used.]

eg) Triggers: cronjob and pollSCM

```
pipeline {
    agent any
    triggers{
        cron ('30 * * * *')
        pollSCM ('* * * * *')
    }
    stages {
        stage('Hello') {
            steps {
                echo “test cronjob”
            }
        }
    }
}
```

17) Parallel execution

- The parallel directive allows both stages to run at the same time.
- **Need of parallel directive:** Compare two or more stages, speed up process.
- Generally, its used to compare two or more stages parallelly.

  eg) if we have source code and we need to check it with sonar and python to validate it so we can give both in 2 different stages after which using parallel execution, we can compare its output at same time as both stages will run parallelly.
- When parallel execution is running then it will wait till all stages under it to get executed after which will move to next stage.

```
Syntax: parallel {
Stage1
Stage2
Stage_n
}
```

eg) Parallel execution

```
pipeline {
  agent any
  stages {
    stage ('parallel execution') {
      parallel {
        stage('sonar-test') {
          steps {
            echo "sonar execution"
            sleep 5
            echo "sonar execution1"
            sleep 7
            echo "sonar execution2"
            sleep 9
            echo "sonar execution3"
            sleep 11
            echo "sonar execution4"
            sleep 5
          }
        }
        stage ('python test') {
          steps {
            echo "python execution"
            sleep 6
            echo "python execution1"
            sleep 8
            echo "python execution2"
            sleep 10
            echo "python execution3"
            sleep 12
          }
```

```
            }
          }
        }
        stage ('test execution') {
          steps {
            echo 'parallel test'
          }
        }
      }
    }
```

18) Post build actions

- Actions that are performed after the build steps are completed.
- Examples include sending notifications, archiving artifacts (for example, zip files so that they can be downloaded later).
- It's used to check status of our pipeline once it's executed.

Types of conditions:

a. **success:** if job/stages get success then this condition will get executed.
b. **failed:** if job/stages get failed then this condition will get executed.
c. **always:** Irrespective of condition (success or failed) this condition will be executed.
d. **changed:** compare with old build status, when any changes (success to failed and vice-versa) with old and current build then this condition get executed.
e. **fixed:** compare with old build from failed(old) to success(current).
f. **aborted:** when any job gets aborted due to any condition then it gets executed.
g. **unstable:** when job gets failed, usually caused by test failures, code violations, etc then it gets executed.
h. **regression:** if old build is success and current build is failed/aborted/unstable then it gets executed.
i. **cleanup:** cleans everything at last after every other post condition has been evaluated, regardless of the Pipeline or stage's status.

```
Syntax: post {
            success {
                echo 'success'
                }
            }
```

eg) Post build actions

```
pipeline {
  agent any
  stages {
    stage('Hello') {
      steps {
        script {
        cleanWs()
        sh "mkdir test"
      }
    }
```

```
    }
  }
    post {
      success {
        echo "for success exection"
      }
      failure {
        echo "for failed exection"
      }
    }
}
```

19) Tools

Add any tools in Jenkins and checking whether installed or not and it's version

Syntax: tools { tool 'tool_name' }

eg) tools { maven 'maven1' }

sh "mvn --version" #prints version

[Add Tool:

Dashboard → manage Jenkins → tools →

eg) maven installations → add maven → name (maven1) → version (choose ant version like 3.9.8) → save]

eg) connecting to tools like maven

```
pipeline {
  agent any
  tools {
    maven 'maven2'
      }
  stages {
    stage('maven') {
      steps {
        sh 'mvn --version'
      }
    }
  }
}
```

20) try/catch

Purpose: Handle exceptions (errors) that occur during pipeline execution.

Structure:

- try block: Contains the code that might throw an exception.
- catch block: Defines how to handle the exception if it occurs.

Benefits: Prevents pipeline failures due to unexpected errors and enables graceful handling of exceptions.

Syntax: try {
sh "mkdir test"
}
catch (Exception e) {
echo 'test folder is present'
}

[Note: To make try/catch run we need to write code in stage or scripts (stage --> steps --> script --> try/catch]

eg) try/catch

```
pipeline {
    agent any
    tools {
        maven 'maven2'
    }
    stages {
        stage('Hello') {
            steps {
                script {
                cleanWs()
                sh 'mvn --version'
                try {
                    sh "mkdir test"
                }
            catch (Exception e) {
                    echo 'test folder is present'
                }
            }
        }
    }
}
```

21) Connecting to folder/directory inside GitHub repository

We can connect any folders that's inside GitHub repository using repository URL syntax and directory syntax.

repo syntax: git branch: 'branch_name', credentialsId: 'created_credential_git',
url: 'GitHub repository URL'

directory syntax: dir('folder_name') {
tasks
}

eg) connecting directory/folder → dir('webapp')

```
pipeline {
   agent any
   stages {
      stage('Hello') {
         steps {
            git branch: 'main', credentialsId: ' githubtoken', url:
'https://github.com/YatishV13/Git_Practise.git'
            sh "pwd"
            sh "ls -lrt"
         }
      }
      stage('test'){
         steps {
            dir('webapp'){
               sh "pwd"
               sh "ls -lrt"
            }
         }
      }
   }
}
```

22) *when Block*

Used within a pipeline stage to conditionally execute steps based on specific conditions.

Structure:

- when block: Defines the condition for step execution.
- Steps within the when block are only executed if the condition is met, other steps within when block will be skipped.

(a) allOf: Execute the stage when all of the conditions are true. Must contain at least one condition. When all condition match then it gets executed.

(b) anyOf: Execute the stage when at least one of the conditions is true. Must contain at least one condition. If one condition matches out of many then it gets executed.

To use when block we need to give parameters and pass it during run, depending on parameters pass we when will run and other will skipped

[Note: When we use when condition then it will execute only that stage which matches with given parameter, remaining stages will be skipped.]

eg1) when block:

```
pipeline {
   agent any
   stages {
```

```
        stage('Hello') {
          when {
            expression {
              branch == 'main'
            }
          }
          steps {
            echo ‘testing when block’
          }
        }
        stage('test'){
          when {
            expression {
              branch == 'test'
            }
          }
          steps {
            dir('webapp'){
              print branch
              // print name
              sh "pwd"
              sh "ls -lrt"
            }
          }
        }
      }
    }
```

eg2) when conditions: (anyOf and allOf)

```
    pipeline {
      agent any
      stages {
        stage('Hello') {
          when {
            allOf{
              expression { branch == 'main' }
              expression { user == 'siddesh' }
            }
          }
          steps {
            echo ‘testing when block’
          }
        }
        stage('test'){
          when {
            anyOf{
              expression { branch == 'test' }
```

```
                expression { user == 'siddesh' }
            }
        }
        steps {
            dir('webapp'){
                print branch
                // print name
                sh "pwd"
                sh "ls -lrt"
            }
        }
    }
  }
}
```

23) webhooks

Function: A mechanism for triggering Jenkins jobs based on events (like push, pull) from external systems. (GitHub)
Benefits: Enables integration between Jenkins and other tools/services (GitHub) for a more automated CI/CD pipeline.

In the context of Jenkins and GitHub, a webhook is a mechanism used by GitHub to notify Jenkins of repository changes.

Steps to link GitHub repository to Jenkins

Step1: Configurations in Jenkins for Job
Create new job (can be freestyle or pipeline project)
Source code management → Git → Repository URL (copy paste GitHub repository URL)
Build Triggers → enable "GitHub hook trigger for GITScm polling"

Step2: Configurations in GitHub
Open repository which we want to link
Go to Settings
webhooks → add webhook → payload URL [Jenkins URL/github-webhook/ **(or)** (http://instanceIP:8080/github-webhook/)] → content type (application/json) → add webhook
For both push & pull to work, we need to enable Generic Webhook Trigger,
To get Generic Webhook Trigger option then we need to install plugin for it,

- **Add Generic Webhook Trigger plugin:**
  Dashboard → manage Jenkins → plugins → available plugins → generic → select generic webhook trigger → install (once installed then we can see this option in build triggers).
  Give any tokens and copy paste that token in webhook payload,
  GitHub → Repository → settings → Webhook → payload → http://instanceIP:8080/github-webhook/invoke?token=dkyutdunfii7ftfbit

[Note:
Once these two steps are done then repository will be linked to job in Jenkins so any commits happen in that repository in GitHub then it auto triggers that job to run in Jenkins.
We can enable any events that can auto-trigger job like push, pull, etc, So, by enabling it only these events get auto triggered otherwise it won't.
We can enable these in webhooks "which events would you like to trigger this webhook → let me select individual events"]

24) Webhooks Path finding using Payload

- If we want to check any details of pull webhook like repository URL, repository name, pull request status, etc, then we need to open payload.
- To open payload → GitHub → linked repo → settings → webhook (Recent deliveries) → click on delivered webhook
- We can copy paste this payload content in "JSON Path Finder" after wch we can get path of many things like repository URL, repository name, pull request status, etc,
- Add Generic Webhook Trigger plugin: Dashboard → manage Jenkins → plugins → available plugins → generic → select generic webhook trigger → install (once installed then we can see this option in build triggers).
- In Jenkins, Job → configure → Generic Webhook Trigger → name(var_name), expression (copy paste path got in JSON Path Finder and enable JSON Path)
  eg) configuration → To print URL in console output
  Variable → url_name
  Expression → $.sender.url (exclude object)
  Print it in console output → print url
  [**Note:** Website to convert to paths from payload → https://jsonformatter.org/json-viewer/83ad3c]

- It's possible to connect multiple repositories using the same URL, so when we commit, it triggers the job here. Previously, we had to configure both the GitHub repository and Jenkins separately. However, now we can use the same link and add it to the webhooks of that repository. You can find the commit path and identify which repository the commit came from in the console output.

25) securing Jenkins

Importance: Crucial to protect your CI/CD pipeline and infrastructure from unauthorized access.
Methods:

- **Role-based Access Control (RBAC):** Define who has permission to perform certain actions in Jenkins (e.g., create jobs, run builds).
- **Security Plugins:** Utilize plugins for additional security features like credential management, two-factor authentication.
- **Regular Updates:** Keep Jenkins and plugins updated to address potential vulnerabilities.