

CI/CD With Jenkins

- To install Jenkins in Ubuntu:
 - ~ You need to install Java because *Jenkins is written in Java*.
 - ~ Its not a native program (like .exe or .bin), rather it's a **.war** file (**Java Web Application Archive**).
 - ~ To run it, you need JVM (Java Virtual Machine), which comes from JDE/JRE.

```
sudo apt update

sudo apt install openjdk-21-jdk -y

sudo wget -O /etc/apt/keyrings/jenkins-keyring.asc \
  https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key

echo "deb [signed-by=/etc/apt/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
```

- ~ **/var/lib/jenkins** is the home directory of Jenkins. You can see this inside **/etc/passwd**
- ~ Inside **/var/lib/jenkins** the jenkins configuration (**config.xml**) file exists.
- ~ After installing jenkins, you can copy the *public IP* of the instance and open in the browser with port *8080* (remember: TCP with port 8080 should be present in the security group attached to the ubuntu instance).
 - ⌚ After opening the browser, it'll show one path where the initial password is present.
 - ⌚ **/var/lib/jenkins/secrets/initialAdminPassword** : In this file the initial password is stored.
- ~ *If you can't open the jenkins ui through browser, then try updating the security inbound rule for TCP 8080 traffic for all IPv4. sometimes, My IP doesn't work.*
- ~ Change the **jenkins url** to a random domain. Otherwise it'll try to access that public ip only. If your instance is rebooted, then the public IP will be changed, and Jenkins will become slow.

➤ **Jobs in Jenkins**

~ **Freestyle Job**

- ❖ In freestyle, everything is configured in the Jenkins UI.
- ❖ **Graphical Jobs.**
- ❖ Each job has a GUI form where you define:
 - * Where to get code (GitHub, SVN, etc.)
 - * Build steps (e.g., mvn clean install, npm build)
 - * Post-build actions (e.g., deploy, send email)
- ❖ **Pros:**
 - * Easy to create (beginner friendly)
 - * Great for simple projects
 - * No need to learn syntax.
- ❖ **Cons:**
 - * Hard to maintain (if there are many jobs, have to edit each of them manually)
 - * Not portable (configs only stay in Jenkins server, not git repo)
 - * Limited flexibility (complex workflows are difficult to manage)
 - * If jenkins crashes, you loose job definitions (unless backed up)

~ **Pipeline As A Code**

- ❖ Instead of configuring Jobs in UI, **Jenkinsfile** is used.
- ❖ Jenkins read the file and runs the pipeline automatically.
- ❖ Written in Groovy based DSL (Domain Specific Language)

➤ **Plugins vs Tools**

~ Simple analogy:

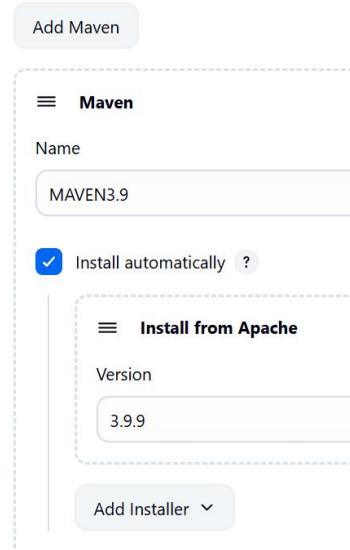
- ❖ Keywords:
 - * Programmer (Jenkins)
 - * Programming Language (Plugin)
 - * Tools (Laptop with compiler installed)
- ❖ If a programmer knows the language (jenkins have plugins installed) but doesn't have a laptop (the server where jenkins present, doesn't have that tool): then it'll be of no use
- ❖ If a programmer doesn't know the language (jenkins don't have the plugin) and he is given a laptop (the server where jenkins is present, have the tools installed): then it'll be of no use

~ **Plugins tell Jenkins how to do things; Tools let Jenkins actually do the work.**

- ~ You can install the tools in the server directly executing the command like **apt install maven ..etc.** OR you can do from the Jenkins GUI as well.

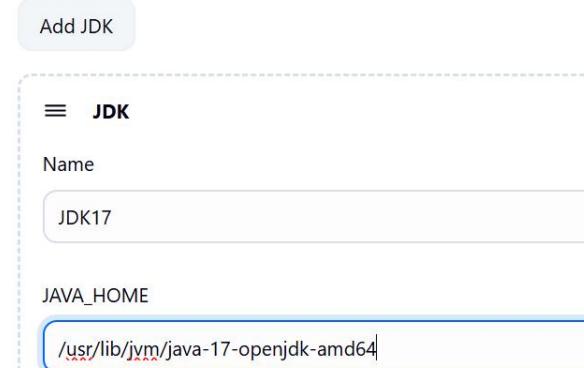
- ~ Note:
 - * In GUI, it'll display only those Tools, whose Plugins are installed.
 - * If you don't see the particular Tool you want, then install its Plugin first.
 - * If you'll install the Tools via system CLI directly; then also it'll be of no use if the Plugin is not installed in Jenkins.

- ~ Ex: I am installing Maven (tool) via GUI



- ~ Its simple, just give a name and select the version.
- ~ Ex-2: I am installing JDK via GUI. Its little different

JDK installations



- ~ Its little different. Installed java-17 version in cli, then gave its home directory path in GUI.

- ~ The tools whose multiple versions can be installed at once in a system (*multiple versions of JDK can be installed in a system*), we need to tell Jenkins that which version is to be used by giving that version's home directory path.
- ~ The installed plugins stay in the directory: `/var/lib/jenkins/plugins`

```
root@ip-172-31-40-120:/var/lib/jenkins/plugins# pwd
/var/lib/jenkins/plugins
root@ip-172-31-40-120:/var/lib/jenkins/plugins# ls
ant config-file-provider github-api.jpi jquery3-a...
ant.jpi config-file-provider.jpi github-branch-source.jpi jquery3-a...
antisamy-markup-formatter credentials github-branch-source.jpi json-api...
antisamy-markup-formatter.jpi credentials-binding.github.jpi json-api...
apache-httpcomponents-client-4-api credentials-binding.jpi gradle.json-path...
apache-httpcomponents-client-4-api.jpi credentials.jpi gradle.jpi json-path...
asm-api dark-theme gson-api.jpi jsoup...
asm-api.jpi dark-theme.jpi gson-api.jpi jsoup.j...
bootstrap5-api display-url-api instance-identity.jpi junit...
bootstrap5-api.jpi display-url-api.jpi instance-identity.jpi junit.j...
bouncycastle-api durable-task ionicons-api.jsp...
bouncycastle-api.jsp durable-task.jspi ionicons-api.jsp...
branch-api echarts-api jackson2-api.jsp...
branch-api.jspi echarts-api.jspi jackson2-api.jsp...
build-timeout edds-api jakarta-activation-api.jsp...
build-timeout.jspi edds-api.jspi jakarta-activation-api.jsp...
caffeine-api email-ext jakarta-mail-api.jsp...
caffeine-api.jspi email-ext.jspi jakarta-mail-api.jsp...
checks-api font-awesome-api javax-activation-api.jsp...
checks-api.jspi font-awesome-api.jspi javax-activation-api.jsp...
cloudbees-folder git jaxb.jsp...
cloudbees-folder.jspi git-client jjwt-api.jsp...
commons-lang3-api git.jspi jjwt-api.jspi mina-ssh...
commons-lang3-api.jspi git.jspi jjwt-api.jspi mina-ssh...
commons-text-api github joda-time-api.jsp...
commons-text-api.jspi github.jspi joda-time-api.jspi nodejs.jsp...
commons-text-api.jspi github.jspi joda-time-api.jspi nodejs.jsp...
```

- ~ All global tools configurations (JDK, Maven, Git, Node.js etc) are stored inside:
`/var/lib/jenkins/hudson.tasks.*`
- ~ Exception: JDKs are stored inside `/var/lib/jenkins/config.xml` because Jenkins treats them as a core runtime tool
- ~ If you have not updated the JDK in Jenkins UI, then you can't see the JDK inside that `config.xml`. And Jenkins will use the default JDK that is present globally (in my case, global default was JDK version 21).

```
root@ip-172-31-40-120:/var/lib/jenkins# cat config.xml | grep -i jdk
<jdks>
  <jdk>
    <name>JDK17</name>
    <home>/usr/lib/jvm/java-17-openjdk-amd64</home>
  </jdk>
</jdks>
```

- ~ If multiples JDKs are configured inside this, then whatever version mentioned in the Job will be used while running the Job inside pipeline.

➤ Lets create our first Job

- ~ Create **FreeStyle** project.
- ~ Give one description like “Learning Jenkins Jobs”
- ~ Skip **Triggers** and **Environments** for now.

- Under **Build Steps**, select **Execute Shell** (the windows part like **execute windows batch commands** will not work as the Jenkins is hosted in Ubuntu in our case).
- Save this now.**

Jenkins / FirstJob

Status: Building (green checkmark)

Changes: Learning Jenkins Job

Workspace

Build Now

Permalinks

- Under the created Job, click on that **Build Now** button 2 or 3 times.

Builds

Filter: /

Today

- #3 12:18PM
- #2 12:18PM
- #1 12:18PM

- You'll see something like this.

Builds

- #3 12:18PM
- #2 12:18PM
- #1 12:18PM

Changes

Console Output

- You can also see the console output of the build.

Console Output

```

Started by user Alok Admin
Running as SYSTEM
Building in workspace /var/lib/jenkins/workspace/FirstJob
[FirstJob] $ /bin/sh -xe /tmp/jenkins8628679474998028620.sh
+ whoami
jenkins
+ pwd
/var/lib/jenkins/workspace/FirstJob
+ w
12:18:40 up 1:02, 3 users, load average: 0.00, 0.00, 0.00
USER     TTY      FROM          LOGIN@    IDLE   JCPU   PCPU WHAT
ubuntu   103.215.237.169  12:11     1:00m  0.00s ?  sshd: ubuntu [priv]
ubuntu   103.215.237.169  11:39     1:00m  0.00s  0.01s sshd: ubuntu [priv]
ubuntu   103.215.237.169  11:17     1:00m  0.00s  0.02s sshd: ubuntu [priv]
+ id
uid=111(jenkins) gid=113(jenkins) groups=113(jenkins)
Finished: SUCCESS

```

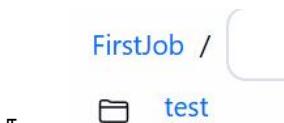
- ↳ You can see the path where the Job ran was
/var/lib/jenkins/workspace/FirstJob
- ↳ You can see some folders inside the path **/var/lib/jenkins**, in which **jobs** and **workspace** are there.
 - ↳ **jobs**
 - * It contains every detail about the job.
 - * Like the build history, configurations, metadata etc.
 - ↳ **workspace**
 - * It is where **Jenkins** actually run build the code and do stuffs.
 - * You can think it like it's a local folder for the **Jenkins user** where it does the things like pulling any repo, building that and testing etc etc.

Workspace of FirstJob on Built-In Node

Status | Changes | **Workspace** | FirstJob /

No files in directory

- * Here there is an option **Workspace**, which remain in sync with the path **/var/lib/jenkins/workspace**.
- * I created one folder inside that path manually using **mkdir** command inside the linux and now it came inside the Jenkins website as well.



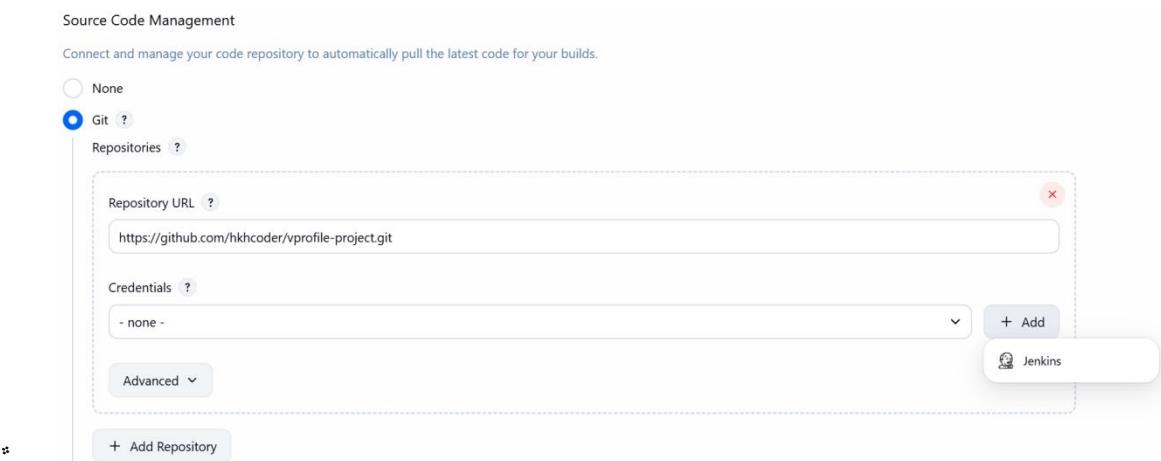
➤ Note

- ↳ The tools that we configure are available globally for all the jobs. Its not bounded to any particular job.
- ↳ Lets suppose JDK, if I have 2 different JDK present inside the tools, then inside the Job, I can select which JDK will be used in my current Job.

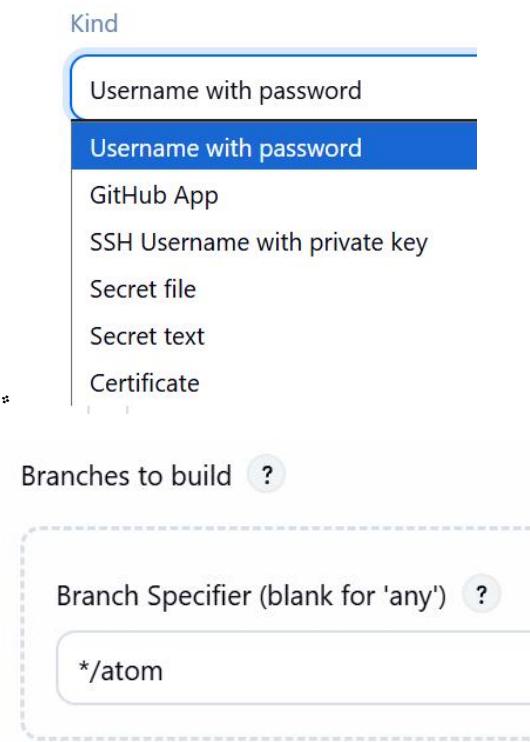
➤ Creating another job to build the vprofile project from github

- ↳ Give a name and description to the job. (it is also **Free Style**).
- ↳ Select the JDK version. (I chose 17)
- ↳ Source Code Management: Choose **Git**.
 - ↳ If the repo is public, then no need to give the credentials.

- Otherwise you need to give clicking on that Add button present in the right.



- You have so many methods using which you can connect to Github.



- Also select the branch from which the code will be build.

- In the previous job, we used **Execution Shell**. But its not recommended.
 - Every time use Plugins to do some specific task.
 - If there is no plugin to do the task you are interested in, then only you should write commands in **Execution Shell**.
 - Here, I chose **Invoke top-level Maven targets**, chose the maven version and the command in the goal i.e. **install** because I want to build the source code.
 - You have some advanced settings as well that you can checkout.

Build Steps

Automate your build process with ordered tasks like:

Invoke top-level Maven targets

Maven Version: Maven3.9

Goals: install

Advanced ▾

Now Lets see the Post-Build Actions

- I chose **Archive the artifacts** and gave ****/*.war** inside the input field *Files to archive*.
 - **** means it'll go to every sub-directory and check if any ***.war** file present and archive that.
- It stores the archived file in somewhere else and give you one link to download or view that. (in the **status** section)

Jenkins / Vprofile Build

Status: Success (green checkmark)

Changes

Workspace

Build Now

Last Successful Artifacts: vprofile-v2.war 79.46 MiB view

IMPORTANT

- When we install any tools from the Jenkins, it install the tool in the Linux (or whatever server where Jenkins is hosted) for the **Jenkins** user only; not **globally**.
- I installed **maven3.9** in the tools section of **Jenkins**.
- Ran one job 2 or 3 times (PS: inside the job under the **invoke top-level Maven targets** the **maven3.9** was selected).
- Then I selected **Default** instead of **maven3.9** in that drop-down and ran built the job again. Now it **failed**.

- ↳ Because, when you choose **default** in that option, it checks **system default maven**, i.e. inside `/usr/bin/mvn` folder which is accessible globally. But maven is not installed in our server globally.
- ↳ So, you need to install **maven** in the **linux server globally** then build the job again. Now it'll **pass**.



- ↳ When you create a new job, at the bottom there is an option **Copy from**, there you can give the name of any existing job you have.
 - ↳ It'll copy all the configs from there to this new job by default.
 - ↳ Means all the fields will be **auto-selected** according to that reference Job.
- ↳ When you install any plugins, then only it'll be visible in the job.
- Just like Gitlab CI/CD, Jenkins also has **environment variables** like **BUILD_ID**, **BUILD_NUMBER** ..etc etc.
- You can use your **own variables** inside the job.

 This project is parameterized ?



- ↳ Inside the configure section, select this checkbox "**This project is parameterized**"
- ↳ Then you'll get the button **Build with Parameters** in place of **Build now**.

 Workspace
▷ Build with Parameters

- ↳ When you click that **Build with parameters** button, you'll get one page where you can enter the values.

Project buildartifact

This build requires parameters:

VERSION

- ↳ Also, you can add the **default value** in that **configure** page.

- Inside the **Manage Jenkins** path, there is an option **System**.
 - ❖ Here you can configure the global configurations. (its not specific to any particular Job)

The screenshot shows the Jenkins 'System' configuration page under 'Manage Jenkins'. The 'Enable BUILD_TIMESTAMP' checkbox is checked. The 'Timezone' dropdown is set to 'Etc/UTC'. The 'Pattern' field contains the value 'ddmmmyy HHmm'. Below the pattern, a note says 'Using timezone: Etc/UTC. Sample time' followed by '(Here I changed the timestamp pattern)'.

```

Enable BUILD_TIMESTAMP ? 
Timezone ? 
Pattern ? 
  Using timezone: Etc/UTC. Sample time (Here I changed the timestamp pattern)

```

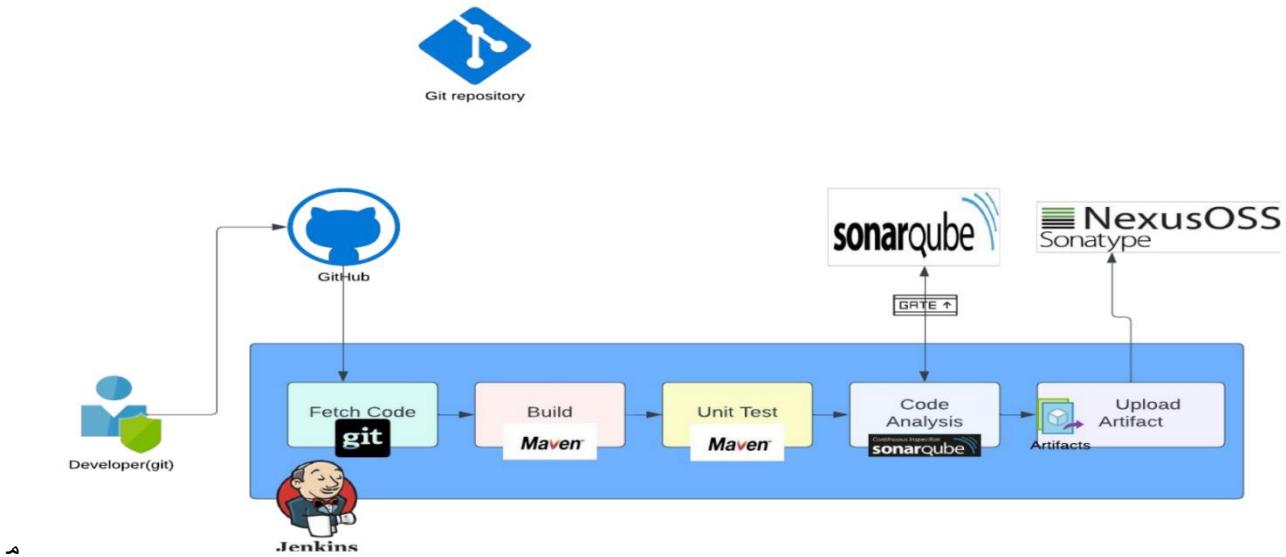
Build Steps:

- `mkdir -p versions`
- `# cp target/vprofile-v2.war versions/vpro$BUILD_ID.war`
- `#cp target/vprofile-v2.war versions/vpro$VERSION.war`
- `cp target/vprofile-v2.war versions/vpro$BUILD_TIMESTAMP.war`

❖ Added in the **execution shell** in **Build Steps**.

- **Disk Space Issue**
 - ❖ Whenever you get any issue for disk space, just increase the volume capacity.

➤ Flow of Continuous Integration Pipeline



↳ SonarQube

- ❖ SonarQube analyzes the source code and generates a report (usually in XML format), which is uploaded to the SonarQube server.
- ❖ Also we can define a Quality Gate — a set of rules (like no critical bugs, minimum 80% test coverage, etc.).
- ❖ If it fails, then pipeline will **stop**.
- ❖ You can think of SonarQube as your automated code reviewer that runs after your build or before deployment to check the quality of your code — not functionality, but cleanliness and security.

↳ Nexus

- ❖ It is you can say an ***artifact repository manager***.
- ❖ It stores built outputs (artifacts) - not source code.
- ❖ Ex: .jar, .war, .zip, .rpm, Docker images, npm packages, Python wheels (.whl)
- ❖ When you execute **mvn clean install**, one .jar file is created inside the **target/** folder. That .jar file is stored inside **nexus repo**, not **github repo**.

- **Steps for Continuous Integration Pipeline**
 - ~ Jenkins setup
 - ~ Nexus setup
 - ~ Sonarqube setup
 - ~ Security group
 - ~ Install necessary plugins in Jenkins (like Nexus, Sonar, Git etc)
 - ~ Integrate
 - ~ Nexus
 - ~ Sonarqube
 - ~ Write pipeline script
 - ~ Set notification
- **Nexus setup**
 - ~ Created an EC2 instance with volume type *t2.medium*.
 - ~ In its security group, allowed **8081** port for Jenkins's sg as it'll be contacted by the port **8081**.
 - ~ Also ssh and **8081** for My IP.
- **SonarQube setup**
 - ~ Created an EC2 instance with volume type *t2.medium*.
 - ~ In its security group, allowed 80 port for Jenkins's sg as it'll be contacted by the port 80.
 - ~ Also ssh and **80** for My IP.
 - ~ SonarQube will contact Jenkins to provide response after the review; and this will be done via port **8080**.
 - ~ So, in the Jenkins SG add SonarQube with port **8080**.
- **NOTES**
 - ~ (all the ports mentioned below is not for the server i.e. EC2 instances, these ports are for the website (jenkins, sonarqube, nexus) hosted on those servers).
 - ~ If an instance is accessible on a particular port (P), and a website is hosted on that same port (P), then any host that connects to the instance via port P will be able to receive responses from that website.
 - ~ means to access the hosted website, first you need to access the instance; then only it'll provide you access to that hosted website.
 - ~ Jenkins is accessible through the port **8080**.
 - ~ **SonarQube**
 - ~ SonarQube's default accessing port is **9000**.

- ↳ We were able to access SonarQube website (hosted in my EC2 server) was because of Nginx setup.

```
server {  
    listen 80;  
    server_name sonarqube.groophy.in;  
  
    location / {  
        proxy_pass http://127.0.0.1:9000;  
    }  
}
```

- ↳ It listens on port **80** and forwards that to **9000**.
- ↳ If you add **9000** port from **My IP** in the Sonar security group, then it can be accessible through port **9000** as well.
- ↳ So, to do proper sharing of information between SonarQube and Jenkins:
 - * Jenkins SG should allow **8080** traffic from Sonar SG.
 - * Sonar SG should allow **80** traffic from Jenkins SG.
- ↳ **nginx doesn't present by default; we had installed that and configured in our code;**

☞ Nexus

- ↳ Nexus runs on port **8081**.
- ↳ So, Sonar SG should allow **8081** traffic from Jenkins SG.



PIPELINE AS A CODE

➤ Introduction

- ~ Automate pipeline setup with Jenkinsfile
- ~ Jenkinsfile defines Stages in CI/CD pipeline.
- ~ Jenkinsfile is a **text** file with Pipeline DSL (domain specific language) syntax.
- ~ Similar to groovy.
- ~ Two Syntax:
 - ~ Scripted
 - ~ Declarative

➤ Syntax (the tree structure of the bullet points represents parent/child/siblings relationship of the commands)

- ~ **pipeline { }**
 - ~ Main block of code.
 - ~ Everything comes inside this **pipeline**.
 - ~ **agent { .. }**
 - ~ Where the job is going to run.
 - ~ **tools { .. }**
 - ~ From the global tools configuration, if you want to include any.
 - ~ For ex: sonar, maven, jdk etc
 - ~ **environment { .. }**
 - ~ Environment variables.
 - ~ **stages { .. }**
 - ~ Steps that will be executed in the Job.
 - ~ **stage { .. }**
 - ~ The syntax will be like **stage("Clone code from git") { .. }**
 - ~ **steps { .. }**
 - ~ Actual commands
 - ~ **post { .. }**
 - ~ Post installation steps.

```

pipeline {
    agent any

    tools {
        maven 'MAVEN3.9'
        jdk 'JDK17'
    }

    stages {
        stage('Fetch code') {
            steps {
                git branch: 'atom', url: 'https://github.com/hkhcoder/vprofile-project.git'
            }
        }

        stage('Unit Test') {
            steps {
                sh 'mvn test'
            }
        }

        stage('Build') {
            steps {
                sh 'mvn install -DskipTests=true' // without -DskipTests it'd run tests again
            }
            post {
                success {
                    echo "Archiving artifact"
                    archiveArtifacts artifacts: "**/*.war"
                }
            }
        }
    }
}

```

~ **tools**

- ~ The names i.e. ‘MAVEN3.9’, ‘JDK17’ should be same as defined in Jenkins global tool configuration.

~ **stage**

- ~ The first word is the plugin (**git** in the provided image)
- ~ And remaining will be the input fields which comes in the UI to enter the values like *branch*, *url*.
- ~ Multiple **stage** can be there.
- ~ Inside the **post** block, there is another block **success**, which will be executed if the pipeline succeeds till that.
 - ~ **archiveArtifacts** is also a plugin.

- ~ Now go to Jenkins website, create one new file and select **Pipeline** instead of **Freestyle**.
- ~ Inside the created Pipeline item, under the **Pipeline** section, there will be 2 options.

Pipeline

Define your Pipeline using Groovy

Definition

Pipeline script from SCM

Pipeline script

Pipeline script from SCM

- * First one if you are pasting the Jenkinsfile code directly there.
- * *Pipeline script from SCM* means if you are taking the code from any repo like **git**. You need to give the url, and path to the Jenkinsfile (mostly its in the root directory only in the name **Jenkinsfile**)
- * In my case, I am going with *Pipeline script*.



- * If you check the pipeline overview, it'll be visible after the build.

- In order to integrate with **SonarQube**, we need to add this in the Tool (as we have installed Sonar Scanner plugin, so there will be an option visible)

☰ SonarQube Scanner

Name
sonar6.2

! Required

Install automatically ?

☰ Install from Maven Central

Version
SonarQube Scanner 6.2.1.4610

- * The exact name (sonar6.2) should be used in the code as well.
- * Now we need to configure the SonarQube server in the **system** page of jenkins.
- * Go to SonarQube >> (click on your profile) >> My Account
 - * And generate one token.

SonarQube installations

List of SonarQube installations

Name
sonarserver

Server URL
Default is http://localhost:9000
http://172.31.21.48:80

Server authentication token
SonarQube authentication token. Mandatory when anor
sonartoken

Advanced ▾ (private ip of sonar server instance is given)

Add Credentials

Domain
Global credentials (unrestricted)

Kind
Secret text

Scope ?
Global (Jenkins, nodes, items, all child items, etc)

Secret
.....

ID ?
sonartoken

Description ?
sonartoken

(token is of type secret text)

* This token will be present inside the page: **manage jenkins > credentials .**

* Now added that token here.

➤ Checkstyle

* It comes in the through a *Maven* plugin called **maven-checkstyle-plugin**.

- ~ When you execute the command **mvn checkstyle:checkstyle** or **mvn checkstyle:check** it'll download the maven plugin *maven-checkstyle-plugin* automatically if it is not there.
- ~ Neither of these commands build the code like **mvn install**.
- ~ **maven checkstyle:checkstyle =>**
 - ↳ This command will generate a report in *xml* file.
 - ↳ The execution of this command doesn't stop even if the validations fails.
 - ↳ It is used to get a report of the code.
- ~ **maven checkstyle:check**
 - ↳ This command doesn't generate a report in *xml* file.
 - ↳ The execution of this command stops if any validation fails.
 - ↳ It is suitable to include in CI/CD. If this command fails, then don't build.
- In our CI/CD code, we'll generate a report using **checkstyle:checkstyle** and upload that to **sonar scanner** to check properly.

```
stage('Checkstyle Analysis') {
    steps {
        sh 'mvn checkstyle:checkstyle'
    }
}
```

- ~ I included this stage in the pipeline.
- ~ **NOTE:** Whatever the execution happens in the pipeline, it'll be stored inside the folder */var/lib/jenkins/workspace/<item name>/*
- ~ So inside that folder, the **xml report** file was generated.

```
stage('Build') {
    steps {
        sh 'mvn install -DskipTests=true' // without -DskipTests it'd run tests again
    }
    post {
        success {
            echo "Archiving artifact"
            archiveArtifacts artifacts: "**/*.war"
        }
    }
}

stage('Unit Test') {
    steps {
        sh 'mvn test'
    }
}
```

- ~ In here **sh** means execute the command in **execution shell** (in free style items we came across this)

```

stage('Checkstyle Analysis') {
    steps {
        sh 'mvn checkstyle:checkstyle'
    }
}

stage("Sonar Code Analysis") {
    environment {
        scannerHome = tool 'sonar6.2'
    }
    steps {
        withSonarQubeEnv('sonarserver') {
            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
            '''
        }
    }
}

```

- ~ That **environment** block can be given in the top level as well (depending upon your usage).
- ~ In my case I only needed this in that specific stage “**Sonar Code Analysis**” so just written the environment inside that stage.
- ~ **withSonarQubeEnv('sonarserver'){ .. }**
 - ~ It is not a normal function call like in Java.
 - ~ Its purpose is to wrap a block of steps and inject environment variables for SonarScanner (SONAR_HOST_URL, SONAR_AUTH_TOKEN, etc.).
 - ~ I gave some **echo** commands to print these default sonar environment variables.

 SONAR_HOST_URL: <http://172.31.21.48:80> >

*  SONAR_AUTH_TOKEN: squ_8677dd75c152fa7cf8

```

stage('Checkstyle Analysis') {
    steps {
        sh 'mvn checkstyle:checkstyle'
    }
}

stage("Sonar Code Analysis") {
    environment {
        scannerHome = tool 'sonar6.2'
    }
    steps {
        withSonarQubeEnv('sonarserver') {
            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
            '''
        }
    }
}

```

- Here the **xml report** is being generated using **checkstyle** and then its uploaded to sonarqube.
- Also **jacoco** is there to test the code coverage.
- After that, in sonarqube the validation will happen.
 - ❖ With the default gate present in sonarqube, the validation will pass for me.
 - ❖ If you want to add custom validation, then you can create custom gate and attach that to the project in sonarqube.



(after the build it'll be created in sonarqube project page)

➤ To create and attach custom gate:

- Click on the link **Quality Gates** in the navigation bar.
- Give one name and create.
- Go inside that newly created **quality gate**, scroll down and click on **Unlock editing** button.

- ~ Click on **Add Condition** button.

Add Condition

On New Code On Overall Code

Quality Gate fails when

Bugs

Operator	Value
is greater than	10

- ~ Now we need to attach this *quality gate* to the project.
- ~ Go inside your project and then:



- ~ Select your created *quality gate*.
 - ~ Now when we run the pipeline again, if the bugs are greater than 10 then the sonar qube validation will fail.
 - ~ **But, the pipeline will still pass as the validation failure occurred in the sonarqube level.**
 - ~ **So we need to return the response from SonarQube to jenkins in another stage, so that jenkins will validate that and fail the pipeline if the desired response is not received.**
 - ~ To achieve this we need to configure **Webhook**.
- **Configuring Webhook**
- ~ When you install **SonarQube** plugin in Jenkins, it automatically exposes a default webhook url <http://<jenkins-url>/sonarqube-webhook/>
 - ~ Go to the **Project Setting** (where the link to attach Quality Gate was there) and click on **Webhooks**.
 - ~ Then Create **Webhook** giving the url as the above format.

Create Webhook

All fields marked with * are required

Name *

vprofile-webhook



URL *

http://172.31.17.119:8080/sonarqube-webhook



Server endpoint that will receive the webhook payload, for example:

"http://my_server/foo". If HTTP Basic authentication is used, HTTPS is

recommended to avoid man in the middle attacks. Example:

"https://myLogin:myPassword@my_server/foo"

Secret

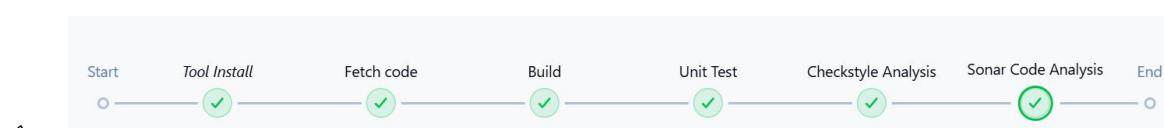


If provided, secret will be used as the key to generate the HMAC hex (lowercase) digest value in the 'X-Sonar-Webhook-HMAC-SHA256' header.

(No need to give Secret)

```
stage("Quality Gate") {
    steps {
        timeout(time: 1, unit: 'MINUTES') {
            waitForQualityGate abortPipeline: true
        }
    }
}
```

- » After configuring **Webhook**, I added this stage in the pipeline at the end.
- » So now, if the validation fails in the sonarqube, it'll send the response to Jenkins so the pipeline will fail.
- » **waitForQualityGate abortPipeline: true**
 - « Here only if the response is for **failure**, then only **abortPipeline: true** will be executed otherwise it'll be skipped.
 - « Also for timeout (if sonarqube doesn't send any response till the desired timeout time)



- « Before adding **Quality Gate** stage.

SonarQube Quality Gate

vprofile-repo Failed

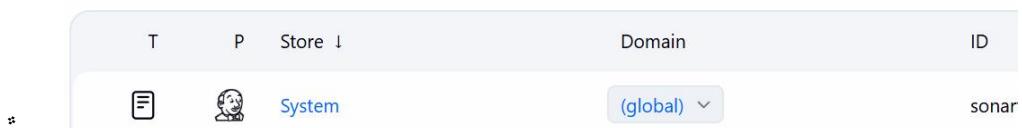
server-side processing: Success

(even if SonarQube failed, pipeline passed)



- ↳ After adding **Quality Gate** stage.
- ↳ **NOTE:** this Webhook will work only if **SonarQube** security group is allowed for port **8080** inside the **Jenkins** security group.
- **Nexus setup (to upload the artifacts to nexus repo)**
 - ↳ first make sure **Nexus Artifact Uploader** plugin is installed in the Jenkins.
 - ↳ First login to Nexus UI and create one repo of type **maven2 hosted** (hosted because we will upload the artifact). **(for downloading: maven2 proxy is used)**
 - ↳ Go to the page: **manage jenkins >> credentials**
 - ↳ Then click on that **global** link

Credentials



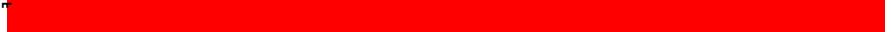
- ↳ Create one credential giving nexus username and password:

[ns](#) / Manage Jenkins / Credentials / System / Global credentials (unrestr... ▾

New credentials

Kind	Username with password
Scope	Global (Jenkins, nodes, items, all child items, etc)
Username	admin
<input type="checkbox"/> Treat username as secret	
Password	*****
ID	nexuslogin
Description	nexuslogin
Create	

NOTE

- “ System defines who can **manage/access** the credentials; Domain defines who (Job/Tool) can **use** those credentials (usually based on URL).
- “ In the URL routing, **system** comes first then **domain**.
- “ Url of system: [...../system/](#)
- “ Url of global domain: [...../system/domain/ /](#)
- “ Url of custom domain: [...../system/domain/aloks.xyz/](#)
- “ So, **system** contains **domains**. And **domains** contains **credentials**.
- “ If you want to add credentials, then its up to you that you'll define that credential for any particular domain or global.
- “ 

- “ In **groovy**, **def** keyword is used to define both **variables** and **methods**.

```

def name = "Alok"
def age = 25

def greetUser(name) {
    // Print to Jenkins console
    echo "Hello, ${name}!"

    // Return a message
    return "Greeting sent to ${name}"
}

// Usage inside a script block
script {
    def result = greetUser("Alok")
    echo "Returned value: ${result}"
}

```

Build Timestamp

Enable BUILD_TIMESTAMP ?

Timezone ?
Etc/UTC

Pattern ?
yy-MM-dd_HH-mm

Using timezone: Etc/UTC; Sample timestamp: 25-10-25_16-35

Export more variables
+ Add

(set this timestamp to use in code)

Rule of Thumb

- ↳ Built-in Jenkins env vars → always use **env.**
- ↳ Variables in environment block → use directly in steps
- ↳ Local def variables → only inside the block, **without env.**

```
stage('Upload Artifact to Nexus') {
    steps {
        script {
            nexusArtifactUploader(
                nexusVersion: 'nexus3',
                protocol: 'http',
                nexusUrl: '172.31.17.55:8081', // private IP of Nexus server
                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 {
            success { echo 'Artifact uploaded to Nexus successfully.' }
            failure { echo 'Failed to upload artifact to Nexus.' }
        }
    }
}
```

- ↳ This is the method provided by that plugin **Nexus Artifact Uploader**.



- ↳ After building that pipeline, it was uploaded.

Notification about success or failure of pipeline

- ↳ I am using **slack notification** plugin.
- ↳ First you need to create one slack account(if not there).
- ↳ I created one channel **devopscid** and added the app to that channel **Jenkins CI**.
(it's a slack app)
- ↳ After adding that app, scroll down in that instruction page and copy the **token**.

Workspace ?

devopsworkspa-a1u5744

Credential ?

slacktoken

Default channel / member id ?

#devopscicd

- ↳ **slacktoken:** you have to add that with that copied token in the previous step
- ↳ Add these things in the **system** page of Jenkins.

```
def COLOR_MAP = [
    'SUCCESS': 'good',
    'FAILURE': 'danger',
    'UNSTABLE': 'warning'
]

pipeline {
```

- ↳ Defined this function at the top of the pipeline code.
- ↳ It'll set the text color based on the Pipeline status.
- ↳ Those keys i.e. **SUCCESS**, **FAILURE**, **UNSTABLE** will be gotten by **currentBuild.currentResult**.

```
def COLOR_MAP = [
    'SUCCESS': 'good',
    'FAILURE': 'danger',
    'UNSTABLE': 'warning'
]

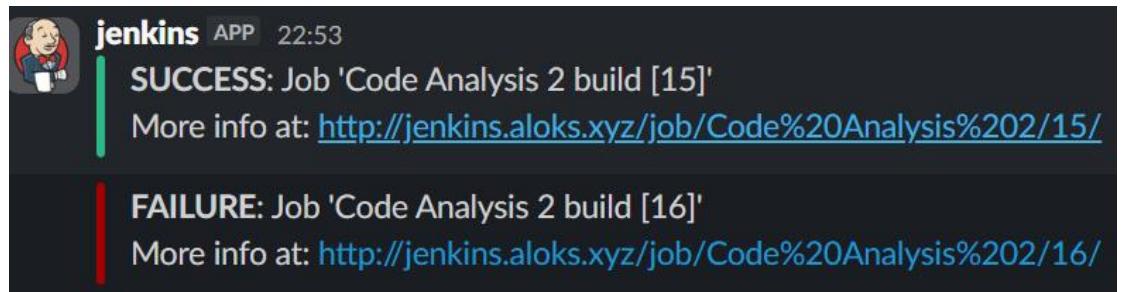
pipeline {
    agent any

    tools { ... }

    stages { ... }

    post {
        always {
            script {
                slackSend(
                    channel: '#devopscicd',
                    color: COLOR_MAP[currentBuild.currentResult],
                    message: "*${currentBuild.currentResult}*: Job '${env.JOB_NAME} build ${env.BUILD_NUMBER}'\nMore info at: ${env.BUILD_URL}"
                )
            }
        }
    }
}
```

- ↳ **Remember: post** block should be written outside the **stages** block otherwise the pipeline will fail even before running.



↳ Here for success **green** and for failure **red** color came in slack.

- D
- F
- F
- F
- D
- F
- D
- F
- D
- F
- F
- D
- F
- D
- F
- D
- F
- F
- D
- F
- F
- F
-