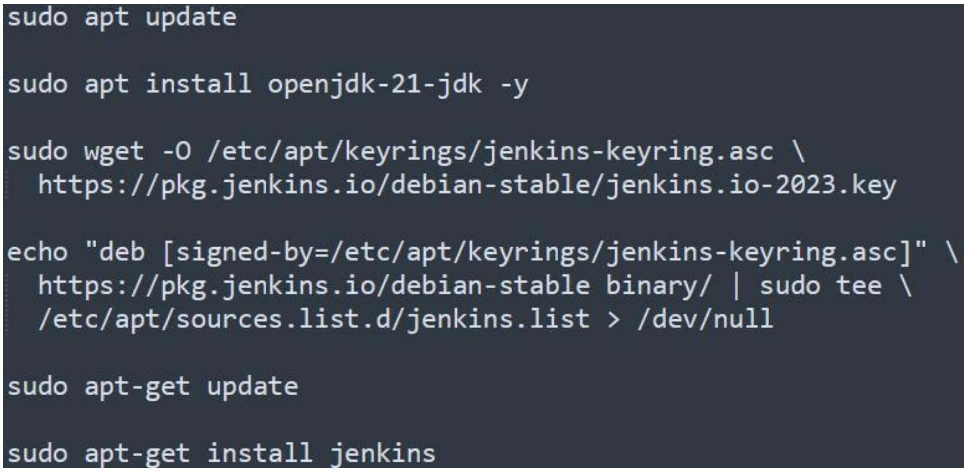
CI/CD With Jenkins

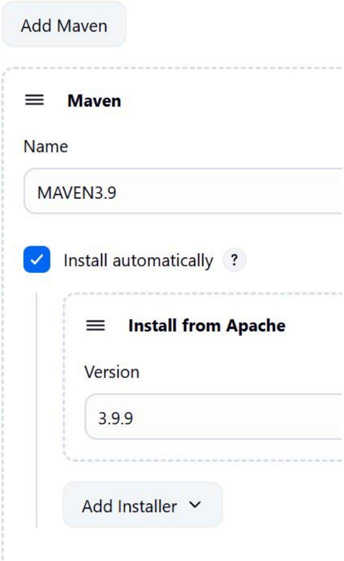
- To install Jenkins in Ubuntu:
  - You need to install Java because *Jenkins is written in Java.*
  - Its not a native program (like .exe or .bin), rather it's a **.war** file (Java Web Application Archive).
  - To run it, you need JVM (Java Virtual Machine), which comes from JDE/JRE.
  - 

```
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

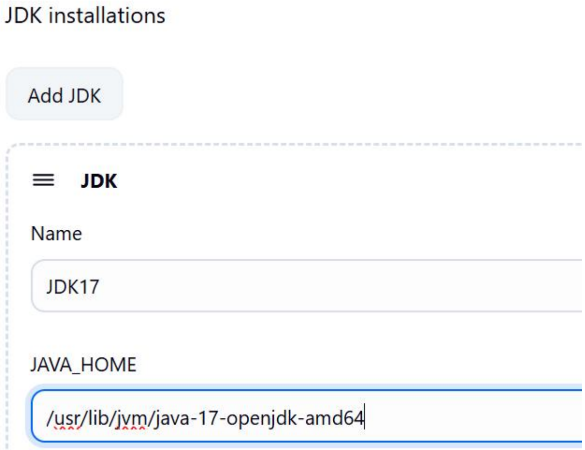


Name: MAVEN3.9

☑ Install automatically ?

Version: 3.9.9

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

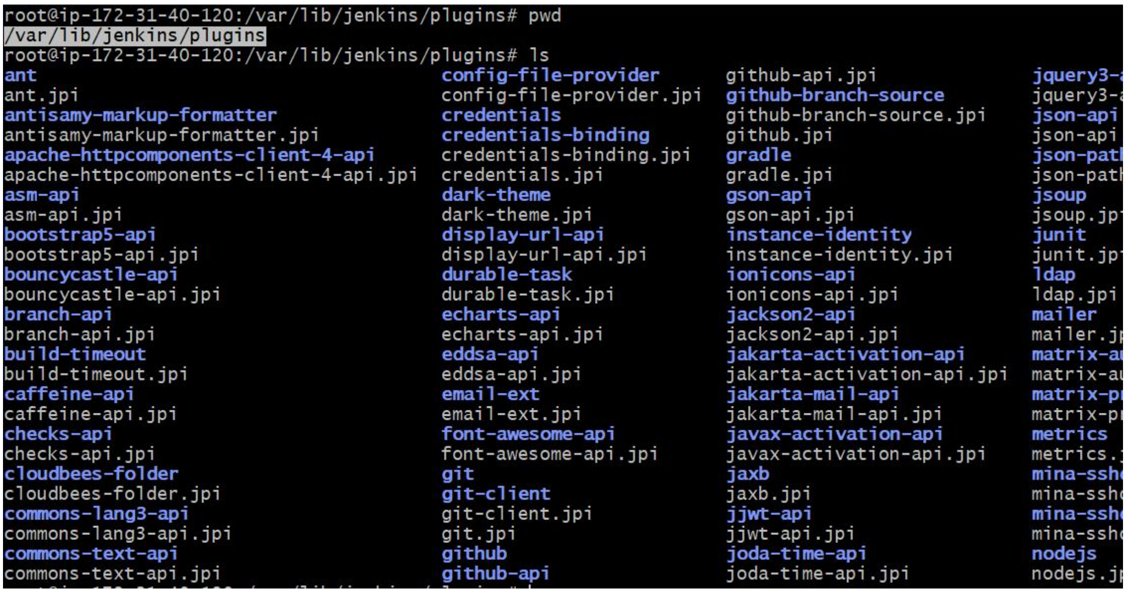


Name: JDK17

JAVA_HOME: /usr/lib/jvm/java-17-openjdk-amd64

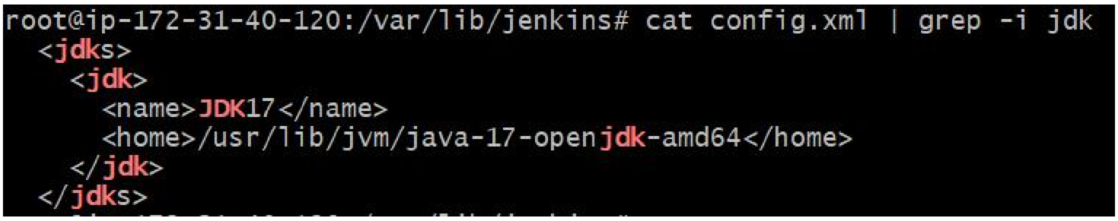
  - Its little different. Installed java-17 version in cli, then gave its home directory path in GUI.

- 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-
ant.jpi                                 config-file-provider.jpi  github-branch-source        jquery3-
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-pat
apache-httpcomponents-client-4-api.jpi  credentials.jpi           gradle.jpi                  json-pat
asm-api                                 dark-theme                gson-api                    jsoup
asm-api.jpi                             dark-theme.jpi            gson-api.jpi                jsoup.jp
bootstrap5-api                          display-url-api           instance-identity           junit
bootstrap5-api.jpi                      display-url-api.jpi       instance-identity.jpi       junit.jp
bouncycastle-api                        durable-task              ionicons-api                ldap
bouncycastle-api.jpi                    durable-task.jpi          ionicons-api.jpi            ldap.jpi
branch-api                              echarts-api               jackson2-api                mailer
branch-api.jpi                          echarts-api.jpi           jackson2-api.jpi            mailer.j
build-timeout                           eddsa-api                 jakarta-activation-api      matrix-a
build-timeout.jpi                       eddsa-api.jpi             jakarta-activation-api.jpi  matrix-a
caffeine-api                            email-ext                 jakarta-mail-api            matrix-p
caffeine-api.jpi                        email-ext.jpi             jakarta-mail-api.jpi        matrix-p
checks-api                              font-awesome-api          javax-activation-api        metrics
checks-api.jpi                          font-awesome-api.jpi      javax-activation-api.jpi    metrics.
cloudbees-folder                        git                       jaxb                        mina-ssh
cloudbees-folder.jpi                    git-client                jaxb.jpi                    mina-ssh
commons-lang3-api                       git-client.jpi            jjwt-api                    mina-ssh
commons-lang3-api.jpi                   git.jpi                   jjwt-api.jpi                mina-ssh
commons-text-api                        github                    joda-time-api               nodejs
commons-text-api.jpi                    github-api                joda-time-api.jpi           nodejs.j
```

- 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).

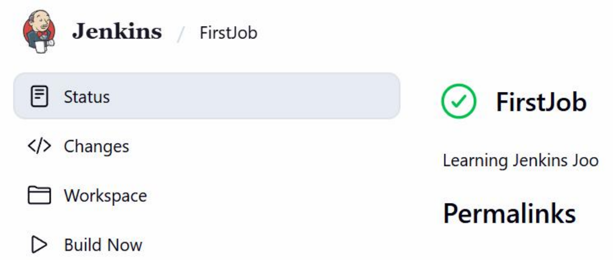
```
root@ip-172-31-40-120:/var/lib/jenkins# cat config.xml | grep -i jdk
  <jdks>
    <jdk>
      <name>JDK17</name>
      <home>/usr/lib/jvm/java-17-openjdk-amd64</home>
    </jdk>
  </jdks>
```

  - 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 out 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.

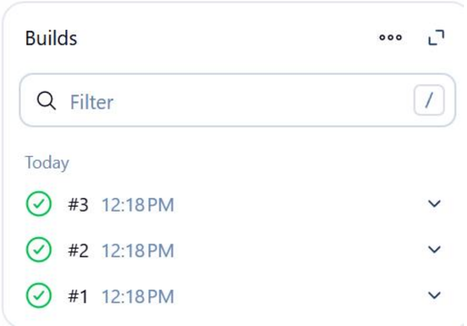


FirstJob

Learning Jenkins Joo

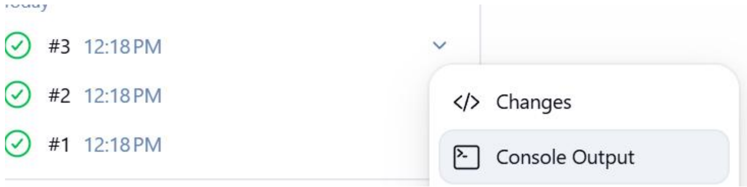
Permalinks

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



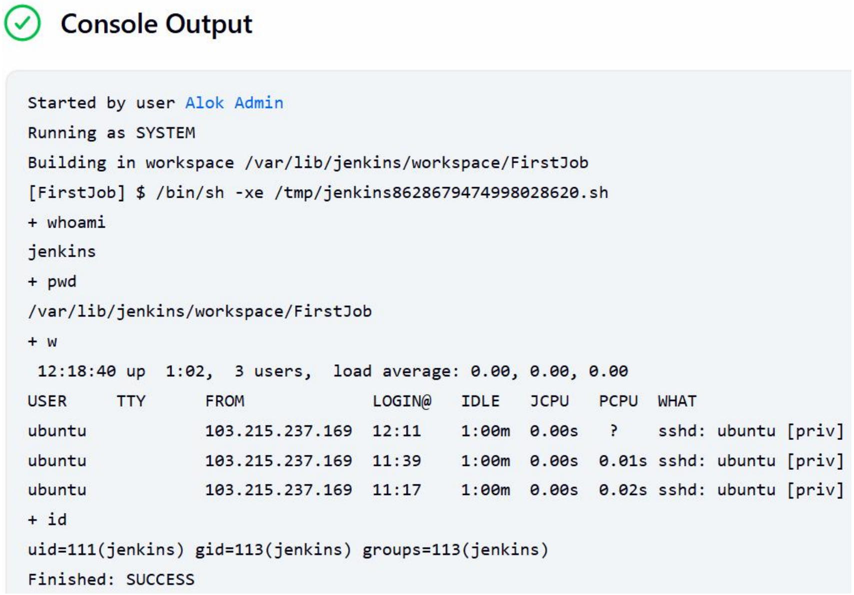
#3	12:18 PM
#2	12:18 PM
#1	12:18 PM

- You'll see something like this.



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

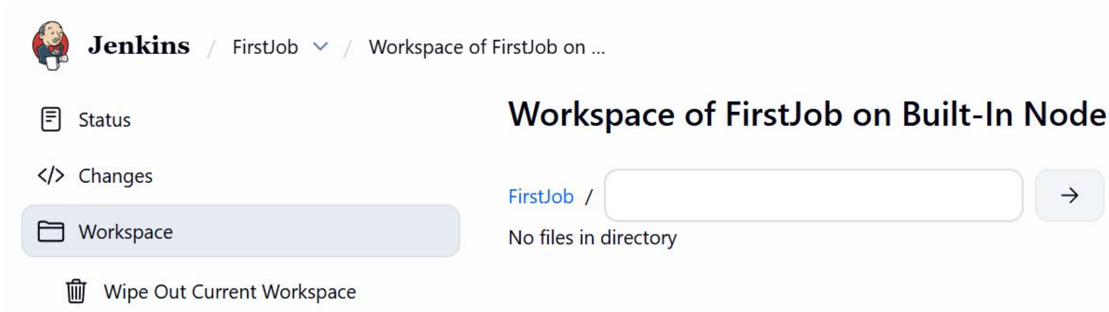
- You can also see the console output of the build.



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
```

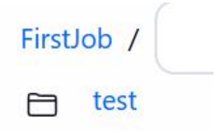
        - You can see the path where the Job ran was **/var/lib/jenkins/workspace/FirstJob**

    - 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

No files in directory

            - Here there is an option W**orkspace**, 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.



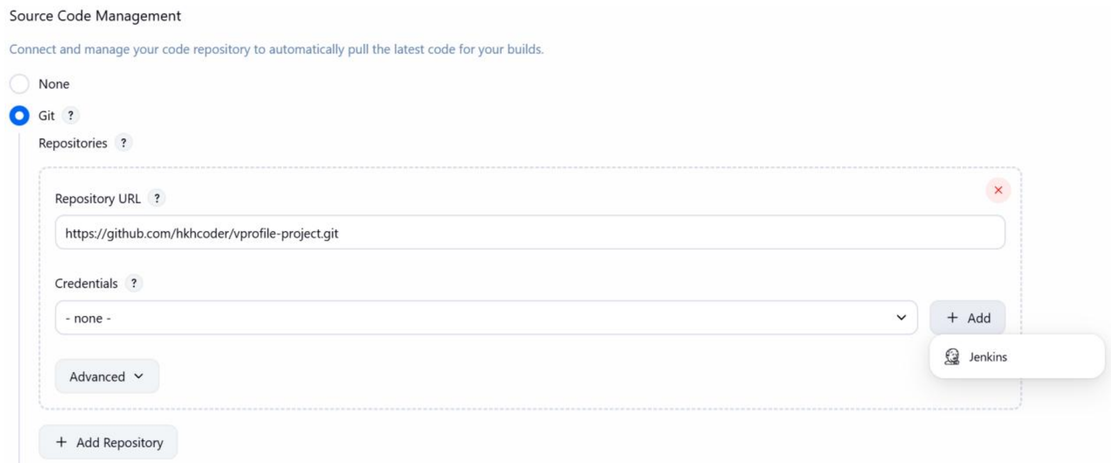
➢ Note

- 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.

      Source Code Management

      Connect and manage your code repository to automatically pull the latest code for your builds.

      ○ None

      ● Git ?

      Repositories ?

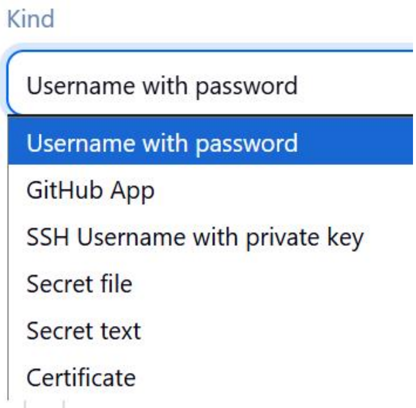
      Repository URL ?

      https://github.com/hkhcoder/vprofile-project.git

      Credentials ?

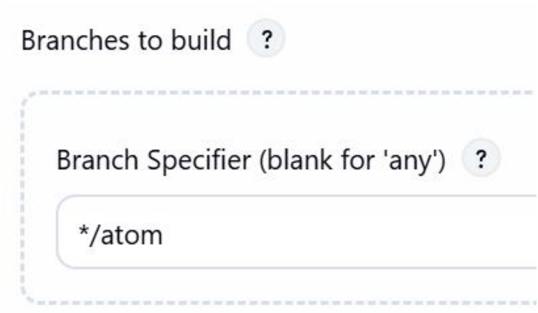
      - none -

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

      Kind

      Username with password

    - 

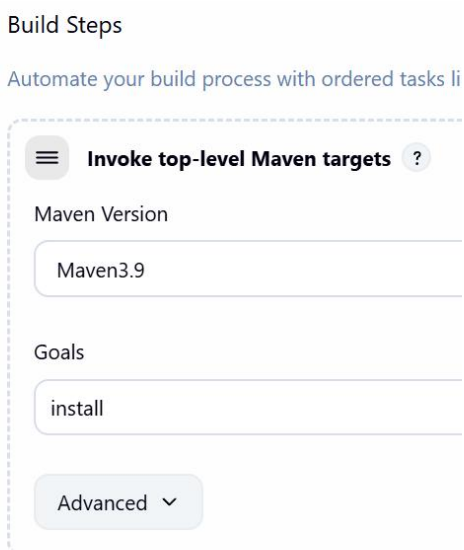
      Branches to build ?

      Branch Specifier (blank for 'any') ?

      */atom

      - 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

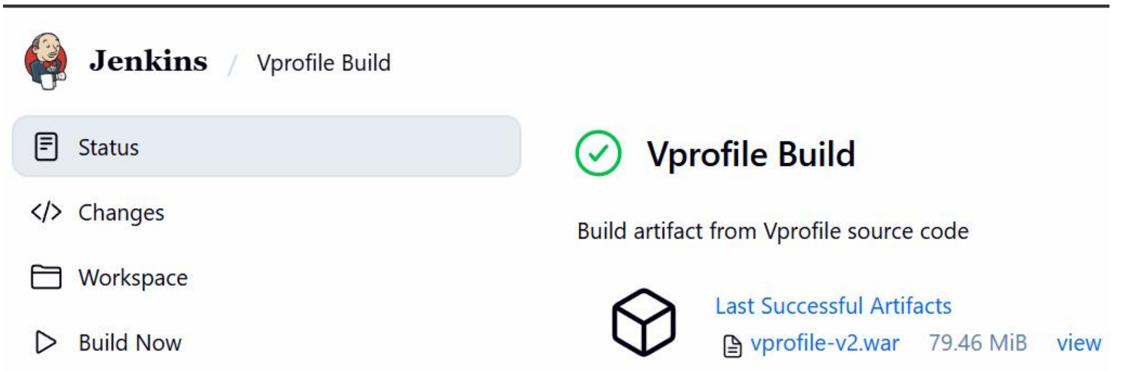
Automate your build process with ordered tasks li

Invoke top-level Maven targets

Maven Version: Maven3.9

Goals: install

- 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)



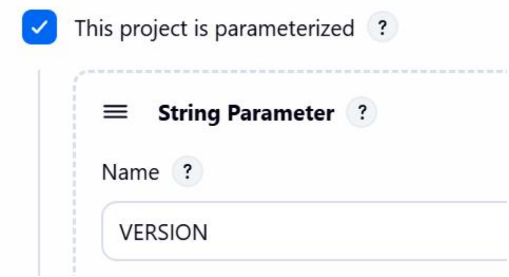
Vprofile Build

Build artifact from Vprofile source code

Last Successful Artifacts

vprofile-v2.war 79.46 MiB

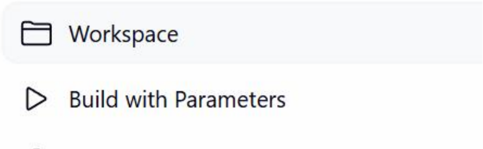
- **IMPORTANT**
  - When we install any tools from the Jenkins, it install the tool in the Linux (or whatever server where Jenkins is hosted) for the **Jenkins** user only; not **globally**.
  - I installed **maven3.9** in the tools section of **Jenkins**.
  - Ran one job 2 or 3 times (PS: inside the 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.

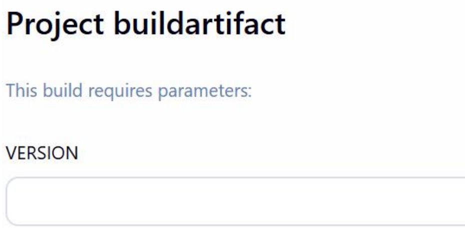
        - Because, when you choose **default** in that option, it checks **system default maven**, i.e. inside **/usr/bin/mvn** folder which is accessible globally. But maven is not installed in our server globally.
        - So, you need to install **maven** in the **linux server *globally*** then build the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

      String Parameter ?

      Name ?

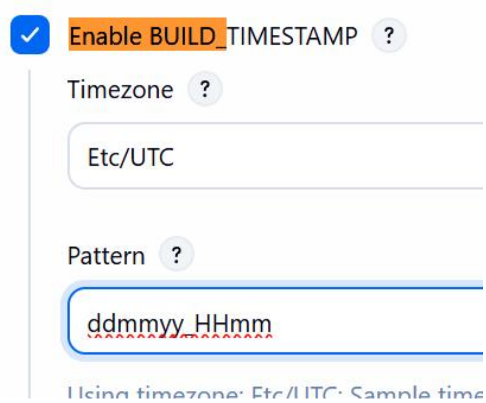
      VERSION
        - Inside the configure section, select this checkbox "**This project is parameterized**"
    - Then you'll get the button **Build with Parameters** in place of **Build now**.
        - 

    - 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)
  - 

    ☑ Enable BUILD_TIMESTAMP ?

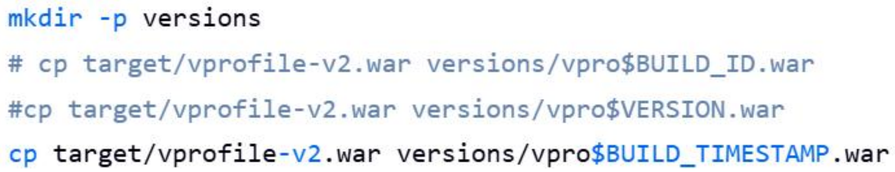
    Timezone ?

    Etc/UTC

    Pattern ?

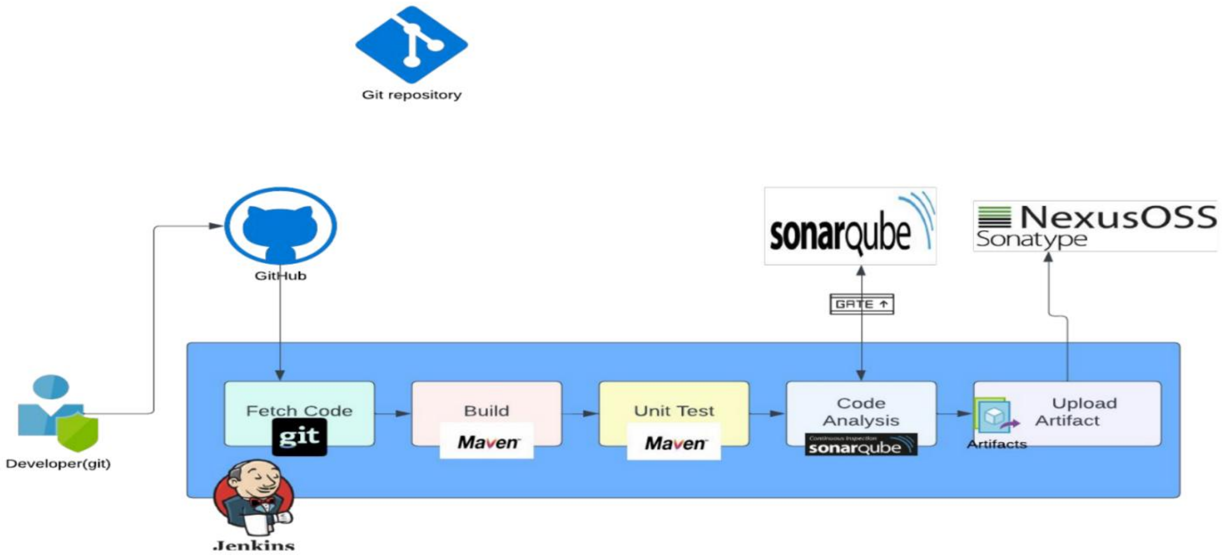
    ddmmyy_HHmm

    Using timezone: Etc/UTC; Sample time

    (Here I changed the timestamp pattern)
  - 

    ```
    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.

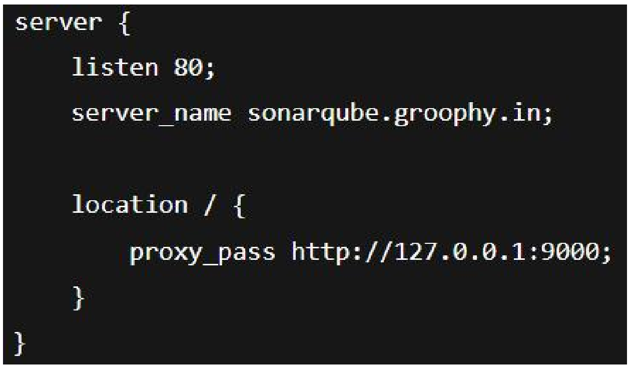
Flow of Continuous Integration Pipeline



- **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
  - Nexux 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.



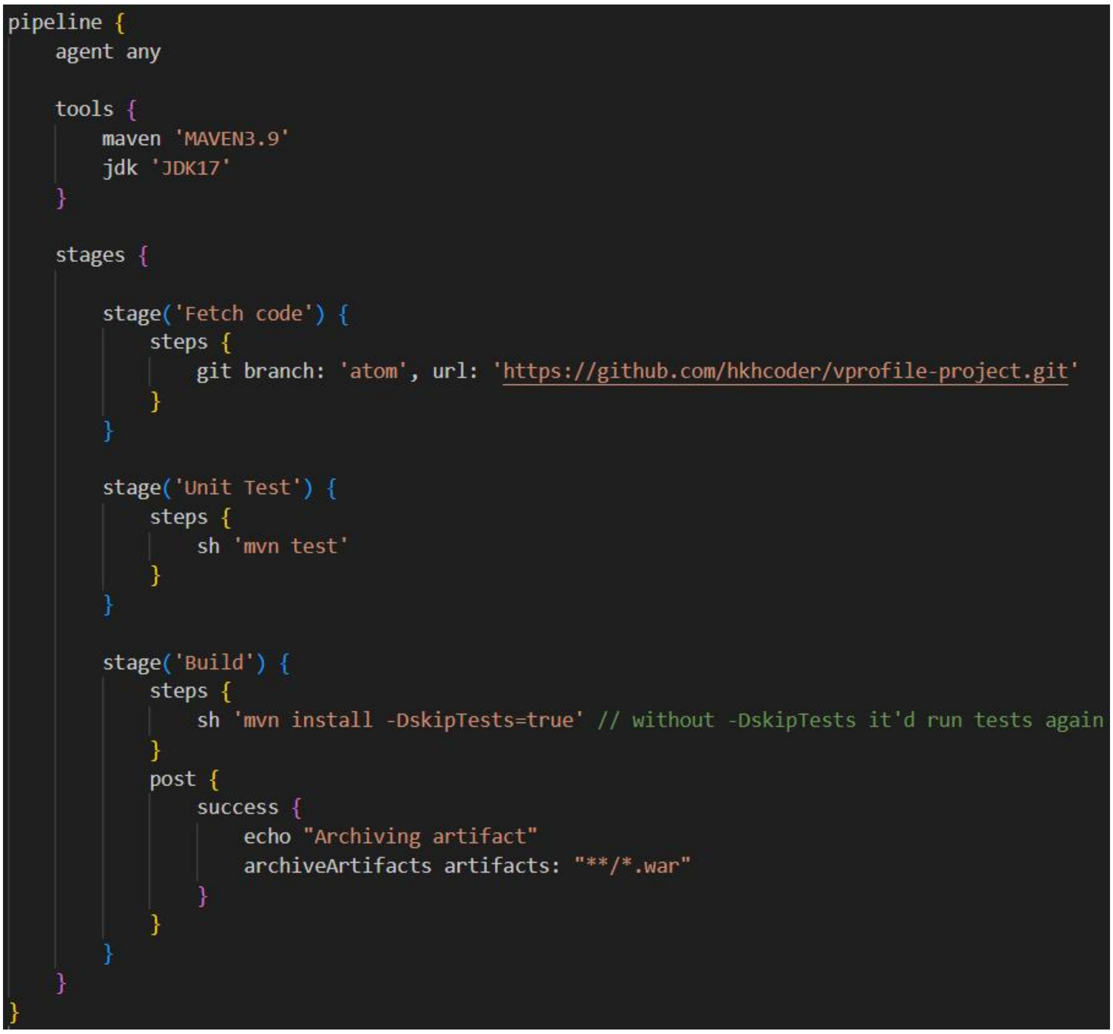
```
server {
    listen 80;
    server_name sonarqube.groophy.in;

    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.

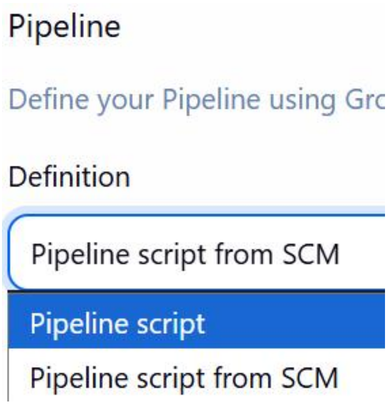
PIPELINE AS A CODE

- **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, it 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.



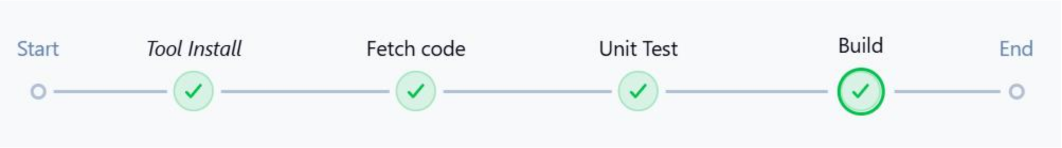
Pipeline

Define your Pipeline using Grc

Definition

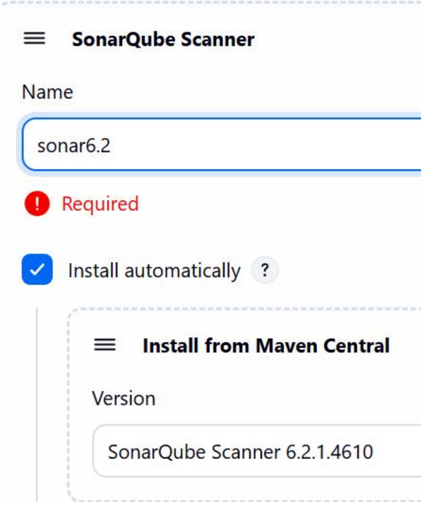
Pipeline script from SCM

- 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 ?

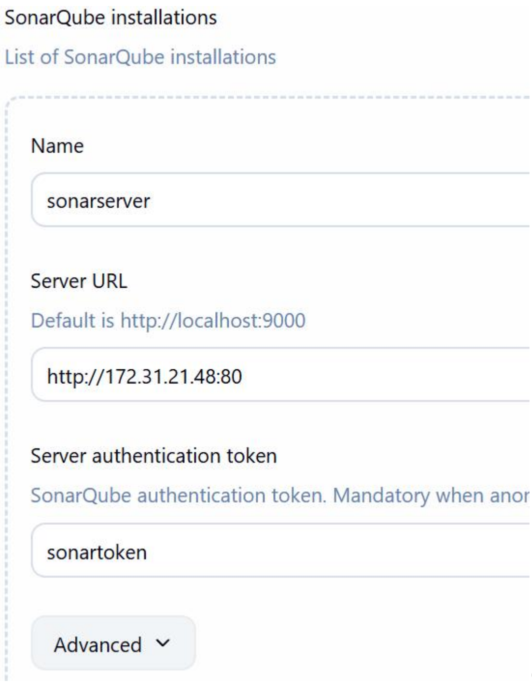
Install from Maven Central

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

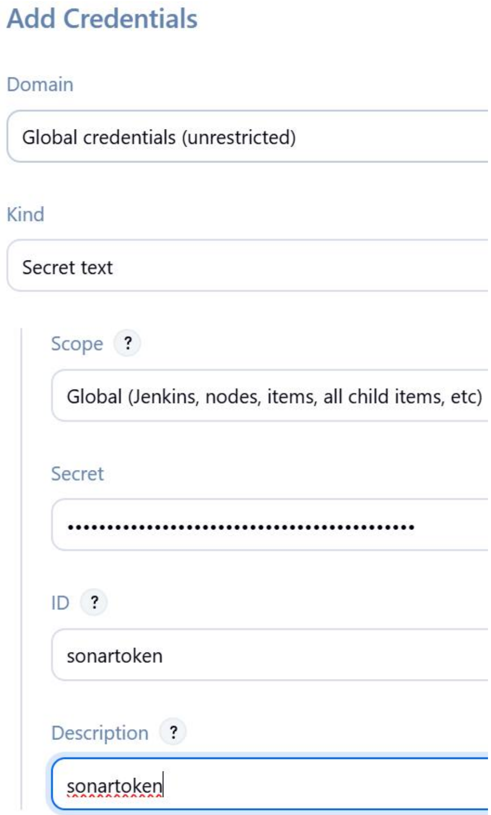
http://172.31.21.48:80

Server authentication token

SonarQube authentication token. Mandatory when anor

sonartoken

- (**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)

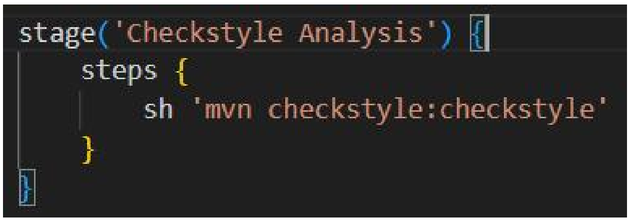
Secret: ••

ID: sonartoken

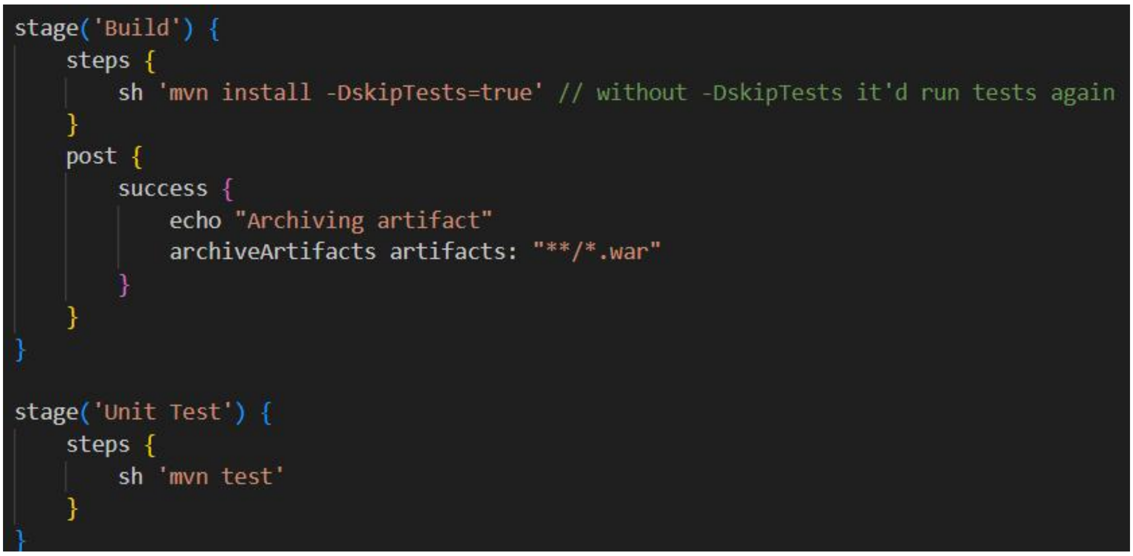
Description: sonartoken

- (token is of type secret text)
  - This token will be present inside the page: **manage jenkins >> credentials .**
- Now added that token here.

- <u>**Checkstyle**</u>
  - It comes in the through a *Maven* plugin called **maven-checkstyle-plugin**.

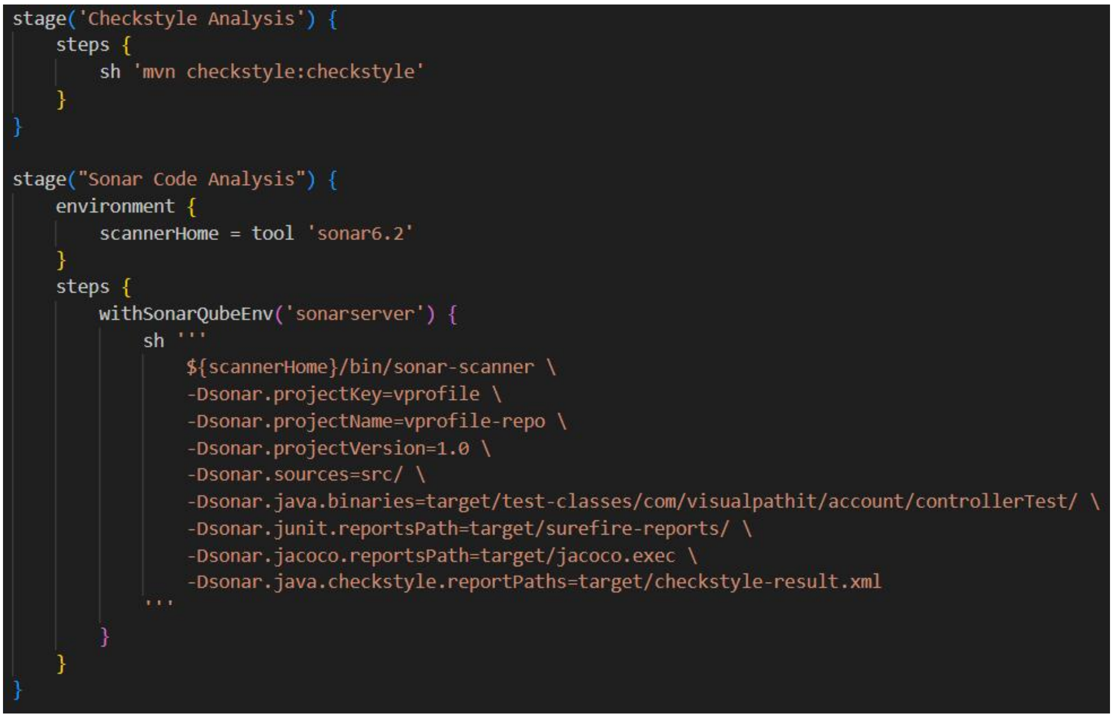
    - When you execute the command **mvn checkstyle:checkstyle** or **mvn checkstyle:check** it'll download the maven plugin *maven-checkstyle-plugin* automatically if it is not there.
    - Neither of these commands build the code like ***mvn install***.
    - **maven checkstyle:checkstyle** =>
        - 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.
- 

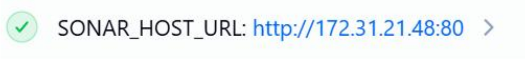
```
stage('Build') {
    steps {
        sh 'mvn install -DskipTests=true' // without -DskipTests it'd run tests again
    }
    post {
        success {
            echo "Archiving artifact"
            archiveArtifacts artifacts: "**/*.war"
        }
    }
}

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)

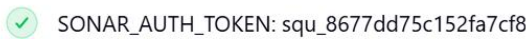


```
stage('Checkstyle Analysis') {
    steps {
        sh 'mvn checkstyle:checkstyle'
    }
}

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
            '''
        }
    }
}
```

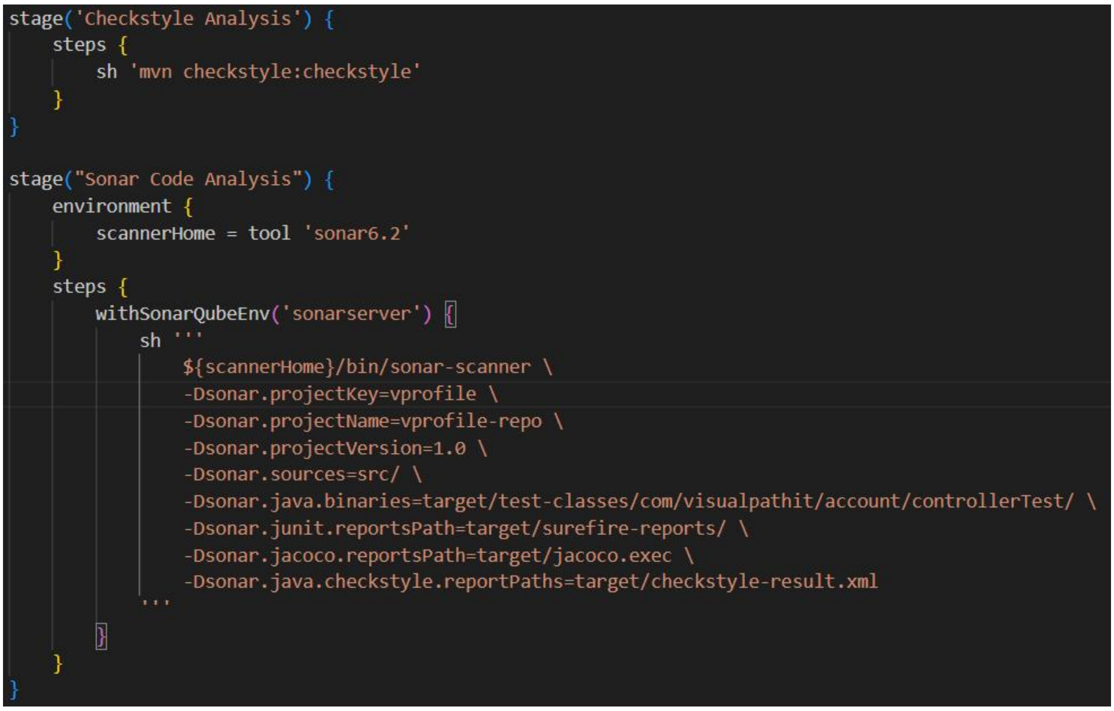
- That **environment** block can be given in the top level as well (depending upon your usage).
- In my case I only needed this in that specific stage **"Sonar Code Analysis"** so just written the environment inside that stage.
- **withSonarQubeEnv('sonarserver'){ .. }**
    - It is not a normal function 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.



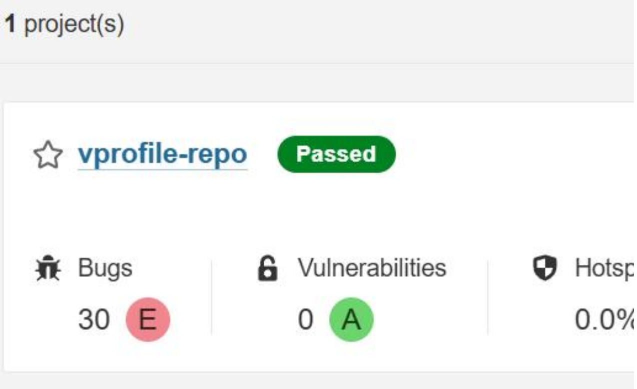
✓ SONAR_HOST_URL: http://172.31.21.48:80



✓ 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
            '''
        }
    }
}
```

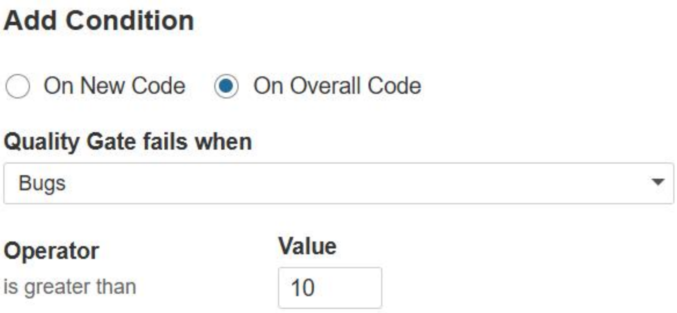
- Here the **xml report** is being generated using **checkstyle** and then its uploaded to sonarqube.
- Also **jacoco** is there to test the code coverage.
- After that, in sonarqube the validation will happen.
  - With the default gate present in sonarqube, the validation will pass for me.
  - If you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

<u>To create and attach custom gate:</u>

- 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

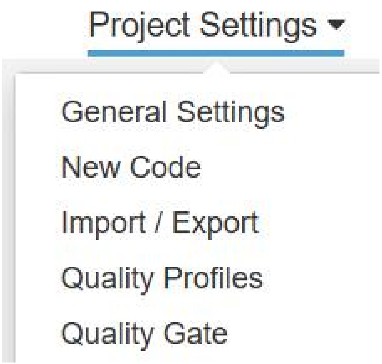
Quality Gate fails when: Bugs

Operator: is greater than

Value: 10

  - Now we need to attach this *quality gate* to the project.

- Go inside your project and then:

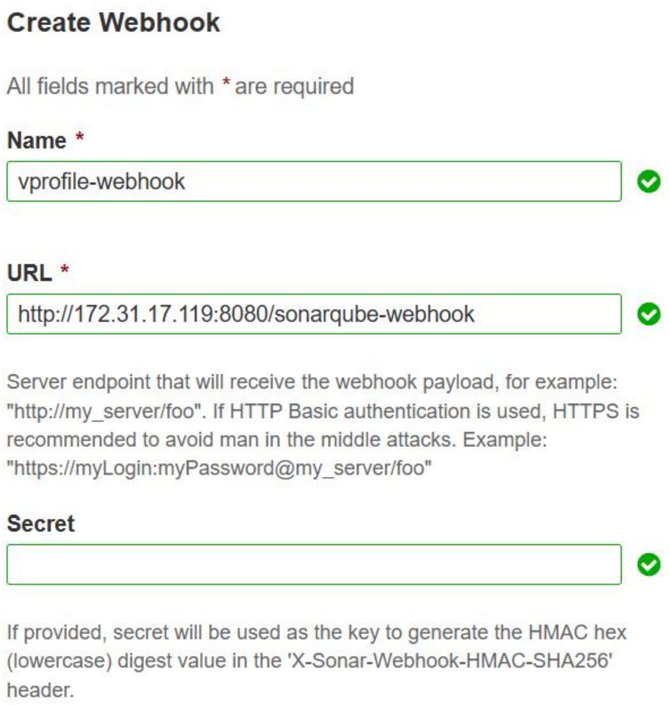


  - click on **Quality Gate**

- 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 achive 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 *

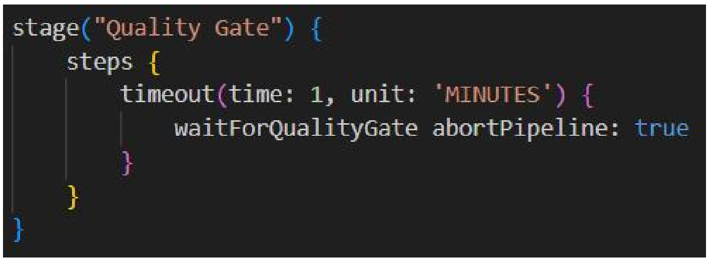
http://172.31.17.119:8080/sonarqube-webhook

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"

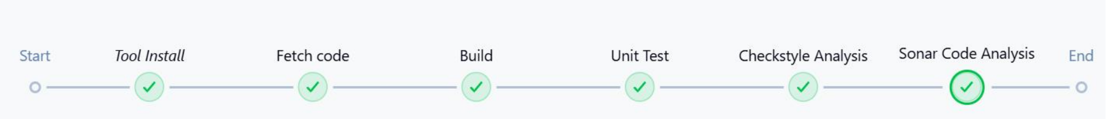
Secret

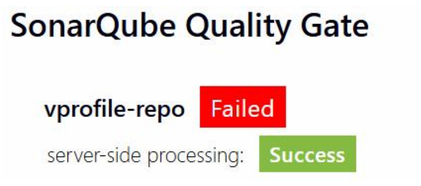
If provided, secret will be used as the key to generate the HMAC hex (lowercase) digest value in the 'X-Sonar-Webhook-HMAC-SHA256' header.

- (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, it 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)
- 

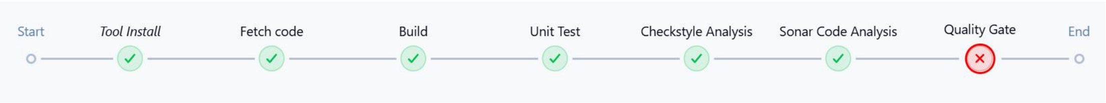
    - Before adding **Quality Gate** stage.
    - 

SonarQube Quality Gate

vprofile-repo Failed

server-side processing: Success

(even if SonarQube failed, pipeline passed)

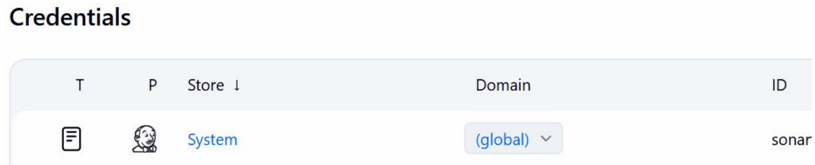


- After adding **Quality Gate** stage.

- **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)

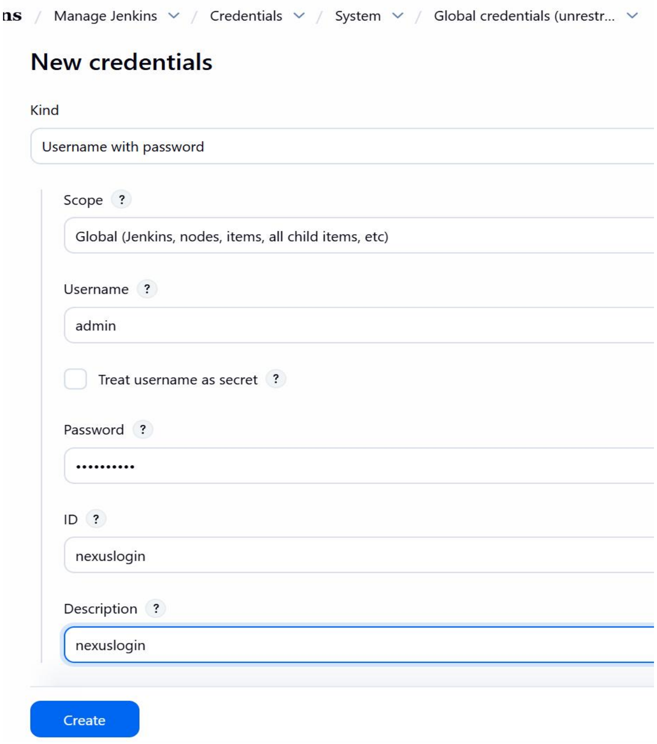
- first make sure **Nexus Artifact Uploader** plugin is installed in the Jenkins.
- First login to Nexus UI and create one repo of type **maven2 hosted** (hosted because we will upload the artifact). (for downloading: **maven2 proxy** is used)
- Go to the page: **manage jenkins >> credentials**
  - Then click on that **global** link



Credentials

T	P	Store ↓	Domain	ID
		System	(global)	sonar

- 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

□ Treat username as secret

Password: ••••••••••

ID: nexuslogin

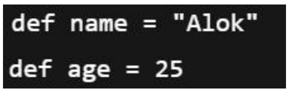
Description: nexuslogin

NOTE

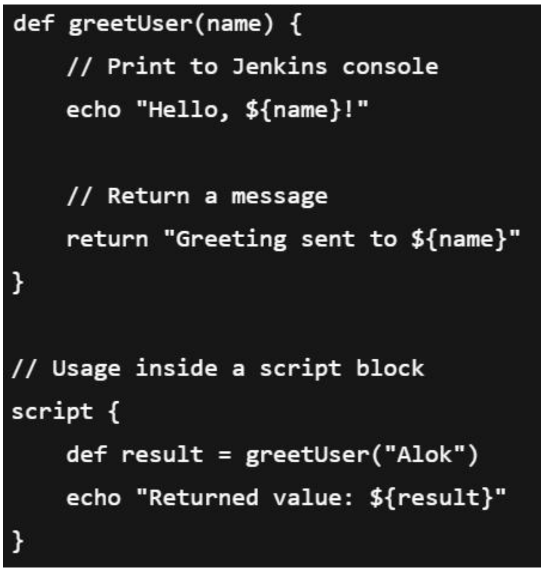
- **System** defines who can ***manage/access*** the credentials; **Domain** defines who (Job/Tool) can ***use*** those credentials (usually based on URL).

- 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.
-

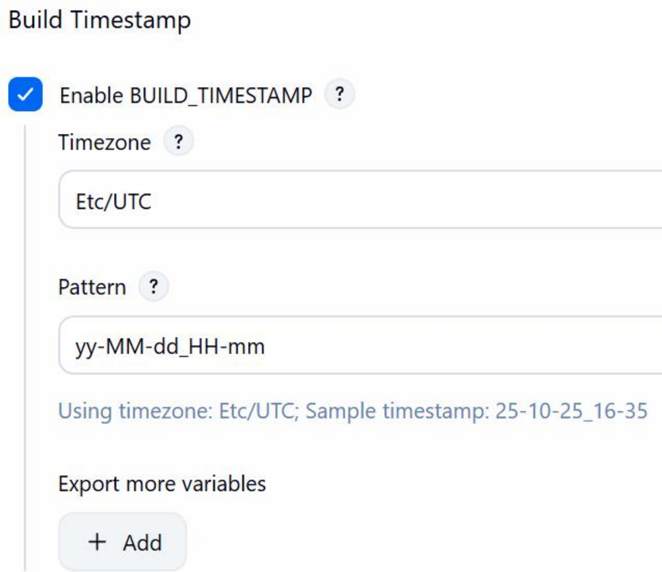
- In groovy, **def** keyword is used to define both **variables** and **methods**.



```
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 ?

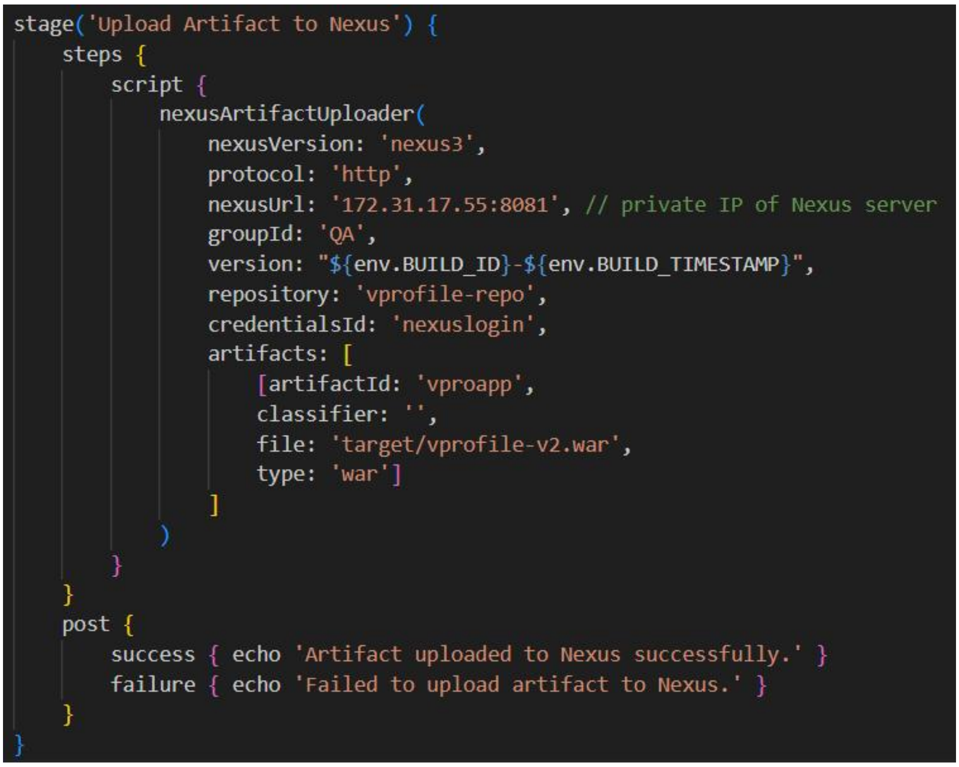
yy-MM-dd_HH-mm

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

Export more variables

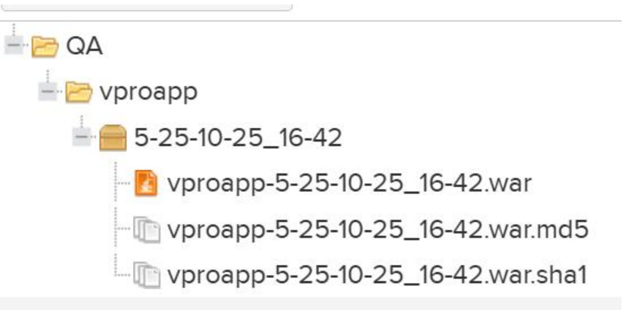
(optional: set this timestamp to use in code)

- **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**.

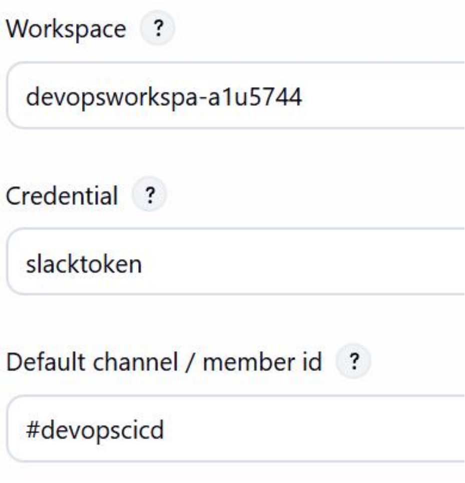
- 

```
QA
  vproapp
    5-25-10-25_16-42
      vproapp-5-25-10-25_16-42.war
      vproapp-5-25-10-25_16-42.war.md5
      vproapp-5-25-10-25_16-42.war.sha1
```

  - After building that pipeline, it was uploaded.

➢ <u>**Notification about success or failure of pipeline**</u>

- I am using **slack notification** plugin.
- First you need to create one slack account(if not there).
- I created one channel **devopscicd** 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

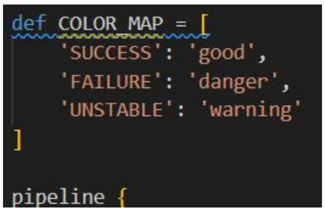
Credential ?

slacktoken

Default channel / member id ?

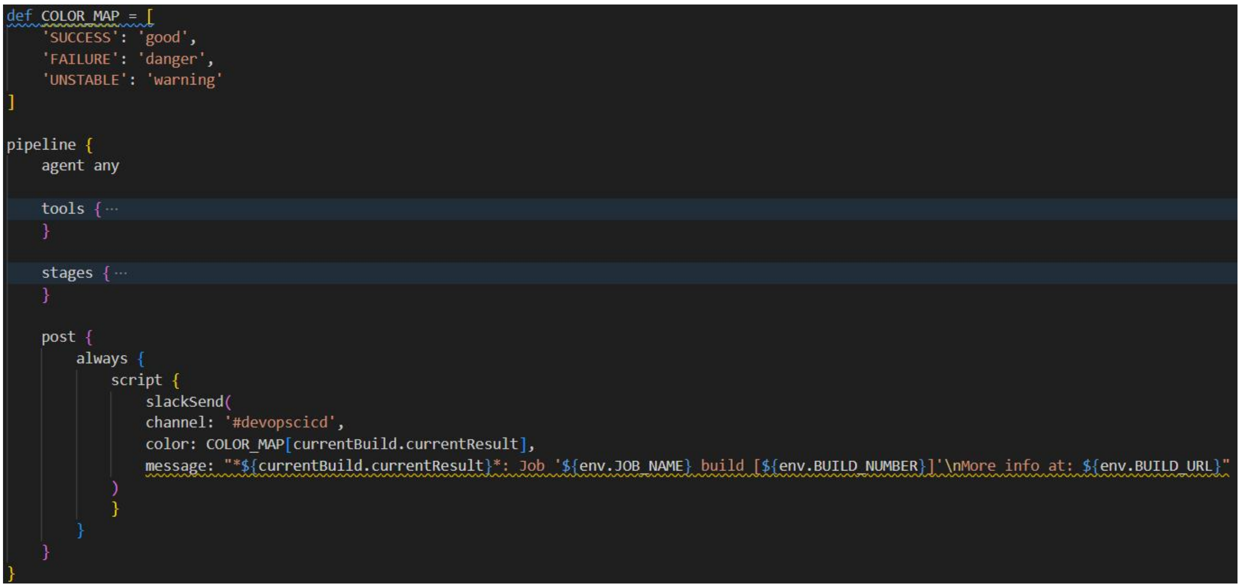
#devopscicd

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



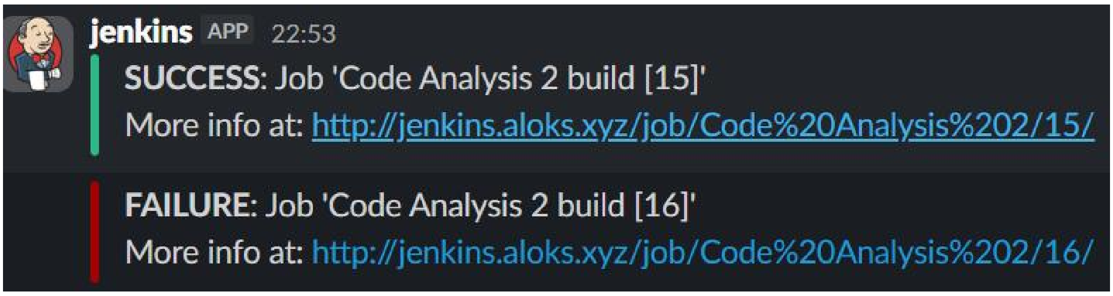
```
def COLOR_MAP = [
    'SUCCESS': 'good',
    'FAILURE': 'danger',
    'UNSTABLE': 'warning'
]

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.



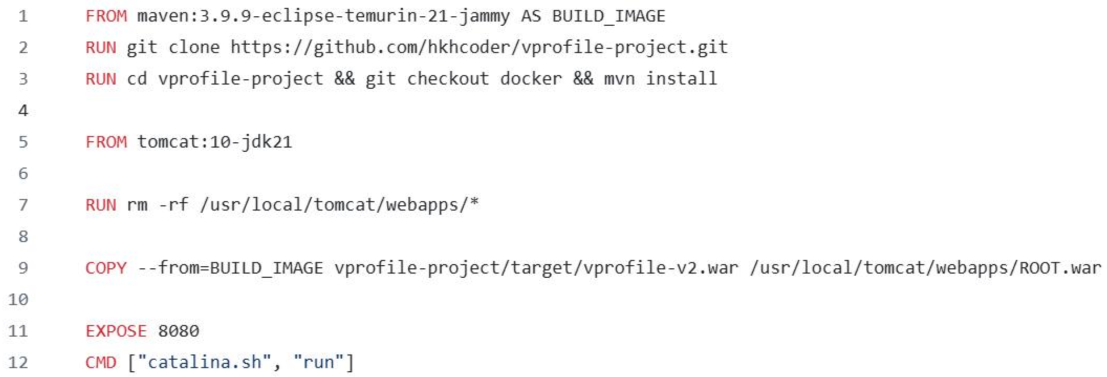
jenkins APP 22:53

SUCCESS: Job 'Code Analysis 2 build [15]'
More info at: http://jenkins.aloks.xyz/job/Code%20Analysis%202/15/

FAILURE: Job 'Code Analysis 2 build [16]'
More info at: http://jenkins.aloks.xyz/job/Code%20Analysis%202/16/

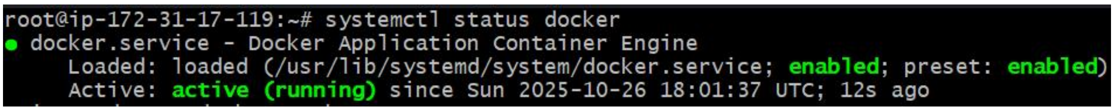
- Here for success **green** and for failure **red** color came in slack.

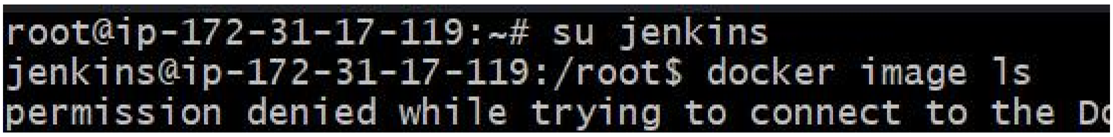
Using Docker



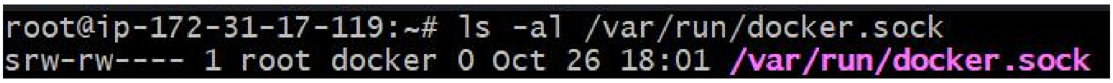
```
1   FROM maven:3.9.9-eclipse-temurin-21-jammy AS BUILD_IMAGE
2   RUN git clone https://github.com/hkhcoder/vprofile-project.git
3   RUN cd vprofile-project && git checkout docker && mvn install
4
5   FROM tomcat:10-jdk21
6
7   RUN rm -rf /usr/local/tomcat/webapps/*
8
9   COPY --from=BUILD_IMAGE vprofile-project/target/vprofile-v2.war /usr/local/tomcat/webapps/ROOT.war
10
11  EXPOSE 8080
12  CMD ["catalina.sh", "run"]
```

- We have this Dockerfile, comprising of 2 steps (line 1 to 3, line 4 to last)
- First it is getting one docker image maven:3.9… and giving alias to that as BUILD_IMAGE.
    - In that it is cloning the repo and building that using **mvn install** command.
- Now its getting the **tomcat** image from dockerhub and hosting that previously built artifact (in the BUILD_IMAGE docker container).

- Steps:
    - AWS => IAM user with Access keys.
    - Install the plugins in Jenkins: *Docker*, *Docker pipeline*, *ecr*, *AWS SDK*
- SSH to Jenkins instance and install aws-cli using the command: **snap install aws-cli --classic**
- Install docker in Jenkins instance. Follow those 2 steps in the docker installation in ubuntu documentation: https://docs.docker.com/engine/install/ubuntu/
    - 

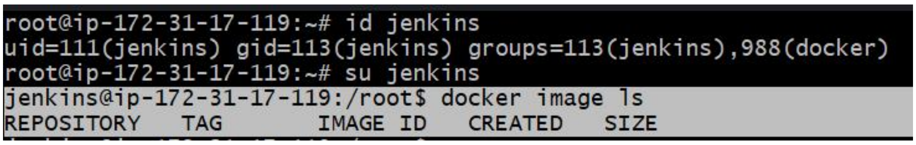
      ```
      root@ip-172-31-17-119:~# systemctl status docker
      ● docker.service - Docker Application Container Engine
           Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; preset: enabled)
           Active: active (running) since Sun 2025-10-26 18:01:37 UTC; 12s ago
      ```
    - If you check now using **systemctl**, the docker should be running.
    - But, only the root user will have the permission to see and run the docker commands.
        - 

          ```
          root@ip-172-31-17-119:~# su jenkins
          jenkins@ip-172-31-17-119:/root$ docker image ls
          permission denied while trying to connect to the D
          ```
        - When checked the list of docker images after logging in with **Jenkins** user, it displayed permission denied.
    - There is a file **/var/run/docker.sock** which is responsible or executing the docker commands.



```
root@ip-172-31-17-119:~# ls -al /var/run/docker.sock
srw-rw---- 1 root docker 0 Oct 26 18:01 /var/run/docker.sock
```

- The user and group has that **read write** access.
- So, we need to add **jenkins** user to the **docker** group.
- **usermod -aG docker jenkins** (it'll add the user **jenkins** to the group **docker**)
    - -a means append. It'll append the group **docker** without overwriting the existing groups.
    - -G means to specify the secondary group.



```
root@ip-172-31-17-119:~# id jenkins
uid=111(jenkins) gid=113(jenkins) groups=113(jenkins),988(docker)
root@ip-172-31-17-119:~# su jenkins
jenkins@ip-172-31-17-119:/root$ docker image ls
REPOSITORY   TAG       IMAGE ID   CREATED   SIZE
```

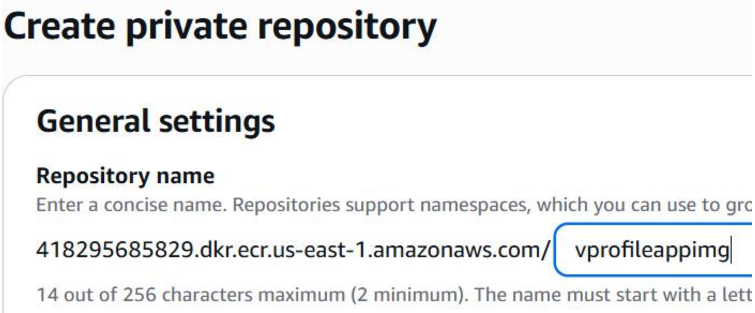
- Now the docker commands can be run with the **jenkins** user.

➤ Now create one IAM user **jenkins** (I gave this name).

- The following access should be granted:
    - AmazonEC2ContainerRegistryFullAccess
    - AmazonECS_FullAccess
- After creating the IAM user, create one Access Key (type CLI).

➤ Now create one **ECR** **(Elastic Container Registry)**

- **ECR** is same as **Docker hub** which is used to store images.
- **ECR** by default host the images in private(but can host pubic as well); similarly **Docker hub** by default host the images in public (but can host private as well).
- Using **ECR** is preferable because it can easily be integrated; no need for extra authentication for **Docker hub**. **IAM** user will handle everything in case of **ECR**.



Create private repository

General settings

Repository name

Enter a concise name. Repositories support namespaces, which you can use to grc

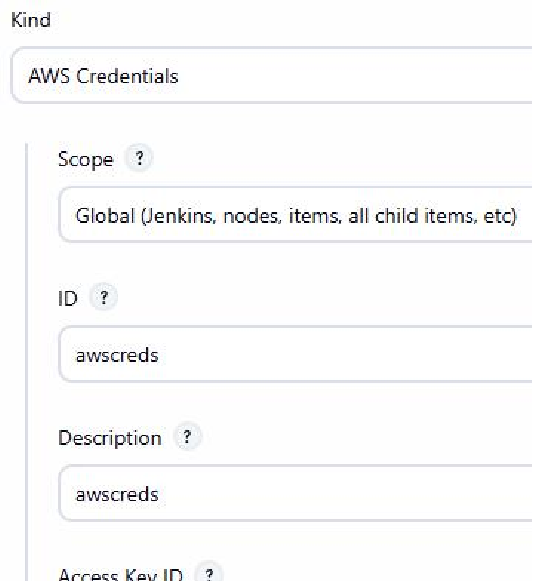
418295685829.dkr.ecr.us-east-1.amazonaws.com/ vprofileappimg

14 out of 256 characters maximum (2 minimum). The name must start with a lett

- Then click on the "**create**" button to create the ECR.

➤ Now install the necessary plugins in Jenkins

- *Amazon Web Services SDK :: All*
- *Amazon ECR*
- *Docker Pipeline* (provides methods like docker.build, docker.withRegistry ..etc)
- *CloudBees Docker Build and Publish* (used in free style mostly)

- Now we need to add **credentials** (Manage Jenkins >> Credentials)
  - You'll get an option **AWS Credentials** in that drop-down where we were selecting the type of credential i.e. username password, secret text etc etc.
  - 

    Kind

    AWS Credentials

    Scope ?

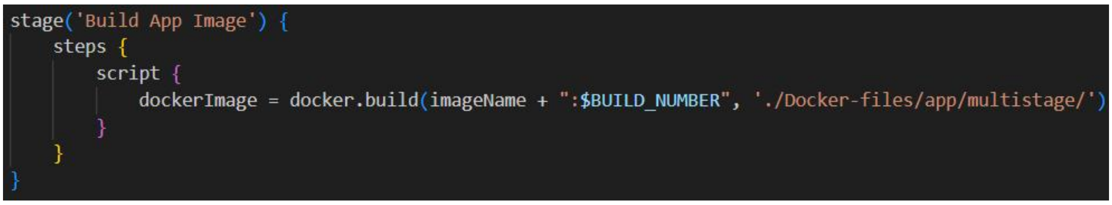
    Global (Jenkins, nodes, items, all child items, etc)

    ID ?

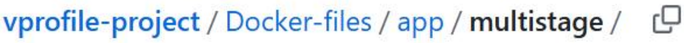
    awscreds

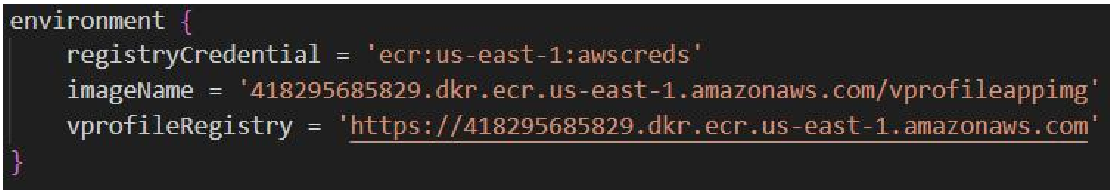
    Description ?

    awscreds

    Access Key ID ?
- Added **Build App Image** stage after the Quality gate stage.
  - We'll remove the stage which was uploading artifacts to nexus, instead we'll build docker images and upload those to the ECR.
  - 

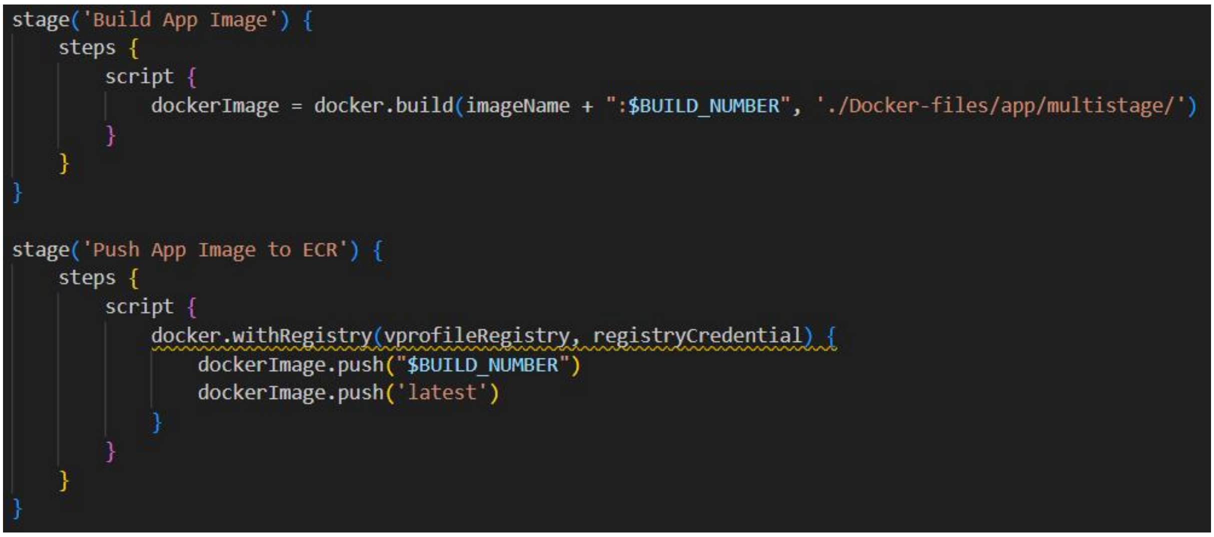
    ```
    stage('Build App Image') {
        steps {
            script {
                dockerImage = docker.build(imageName + ":$BUILD_NUMBER", './Docker-files/app/multistage/')
            }
        }
    }
    ```

    - 2nd argument is the path to the **Dockerfile** in our workspace.
    - In github repo, Dockerfile is present at that path.
      - 

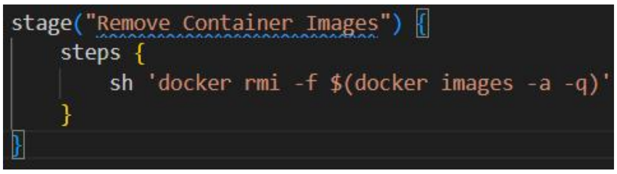
        vprofile-project / Docker-files / app / **multistage** /
- So now below is the changes:
  - 

    ```
    environment {
        registryCredential = 'ecr:us-east-1:awscreds'
        imageName = '418295685829.dkr.ecr.us-east-1.amazonaws.com/vprofileappimg'
        vprofileRegistry = 'https://418295685829.dkr.ecr.us-east-1.amazonaws.com'
    }
    ```

    - That format of registryCredential should be same:
      - **ecr:<region of ecr>:<name of aws credential>**

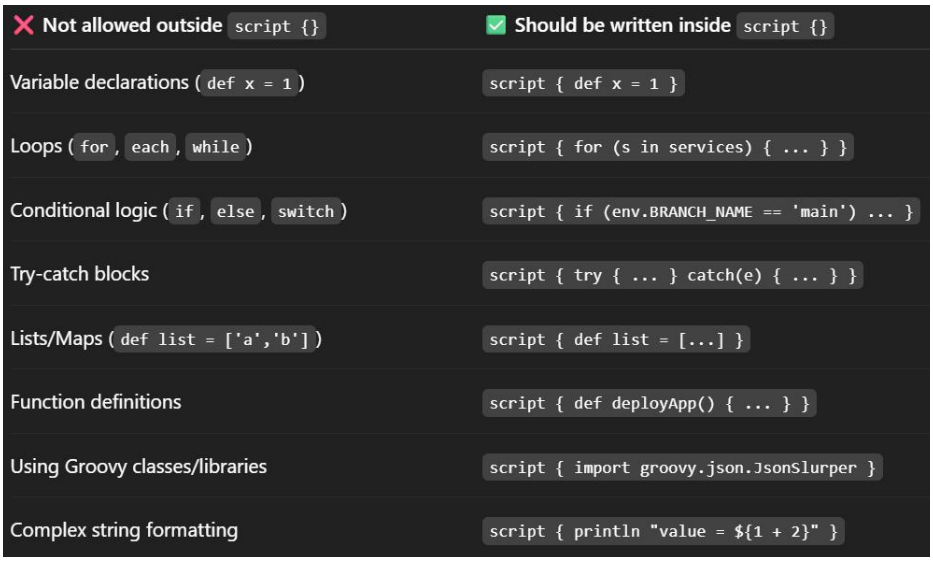


```
stage('Build App Image') {
    steps {
        script {
            dockerImage = docker.build(imageName + ":$BUILD_NUMBER", './Docker-files/app/multistage/')
        }
    }
}

stage('Push App Image to ECR') {
    steps {
        script {
            docker.withRegistry(vprofileRegistry, registryCredential) {
                dockerImage.push("$BUILD_NUMBER")
                dockerImage.push('latest')
            }
        }
    }
}
```

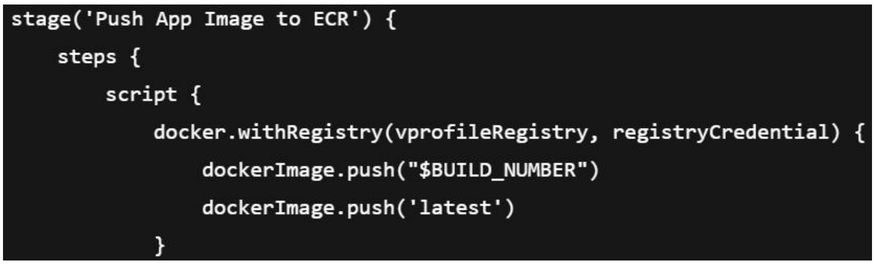


```
stage("Remove Container Images") {
    steps {
        sh 'docker rmi -f $(docker images -a -q)'
    }
}
```

- It is at the end.
- It'll find all the docker images on the host where docker was running (i.e. Jenkins instance) and delete all those images from that machine.
- Images will not be deleted from ECR (Remember).



❌ Not allowed outside `script {}`	✅ Should be written inside `script {}`
Variable declarations (`def x = 1`)	`script { def x = 1 }`
Loops (`for`, `each`, `while`)	`script { for (s in services) { ... } }`
Conditional logic (`if`, `else`, `switch`)	`script { if (env.BRANCH_NAME == 'main') ... }`
Try-catch blocks	`script { try { ... } catch(e) { ... } }`
Lists/Maps (`def list = ['a','b']`)	`script { def list = [...] }`
Function definitions	`script { def deployApp() { ... } }`
Using Groovy classes/libraries	`script { import groovy.json.JsonSlurper }`
Complex string formatting	`script { println "value = ${1 + 2}" }`



```
stage('Push App Image to ECR') {
    steps {
        script {
            docker.withRegistry(vprofileRegistry, registryCredential) {
                dockerImage.push("$BUILD_NUMBER")
                dockerImage.push('latest')
            }
```

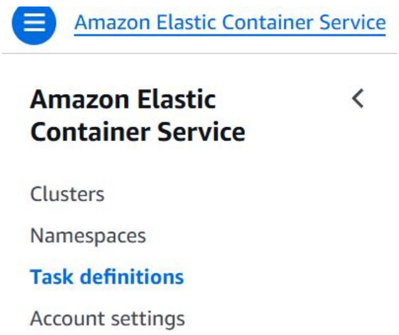
- Here **script** block is not required as we've not used any groovy specific things.

Docker CICD

- We'll be using **ECS** (Elastic Container Service). (alternative of Kubernetes in AWS)
  - We'll have to create 2 things: **cluster**, **service**
  - **Service** is inside the **cluster**, which(service) will fetch the container from **ECR** and run it.
  - We'll be using **withAWS** which is available inside the plugin **Pipeline: AWS Steps**
- **NOTE**
  - You can imagine **cluster** as a **pc**, **task definition** is a programming code that tells **how and what to run**, **service** is something that ensures **n copies of that code is running always**.
  - **Cluster:**
    - A logical group or environment that *organizes and manages all the compute resources* (like EC2 instances or Fargate tasks) where your containers actually run..
    - During creation of cluster you can either choose **EC2** or **Fargate**.
    - If you choose Fargate then you don't need to worry about the instances or group of instances where the containers will be running.
    - It'll scale by itself.
    - AWS automatically allocate resources that'll enough for your container to run.
  - **Task definition:**
    - It defines which Docker image to use.
    - Its kind of a blueprint that defines how to run the container.
    - It describes cpu utilization, port mapping, env variables, IAM role etc.
    - When ECS creates a task, it'll see the *Task Definition* and follow its instruction to spin up the containers.
  - **Service:**
    - It ensures that a specified number of tasks (containers) are always running.
    - It handles *Load Balancing*.
    - Supports *Auto Scaling*.
- Creating ECS cluster and service:
  - Create one cluster after going to the ECS page.
  - Give one name and its better to give one tag as well.

- Then click on create; if it shows any error then create again; it'll succeed.

- Now create one **Task Definition**



Amazon Elastic Container Service

Amazon Elastic Container Service

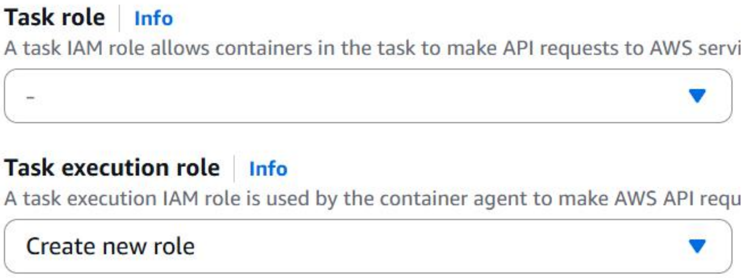
Clusters

Namespaces

Task definitions

Account settings

- I gave the name **vprofileapptask**, launch type **AWS Fargate**, OS **Linux/X86_64**, memory **2gb**,



Task role | Info

A task IAM role allows containers in the task to make API requests to AWS servi

- ▼

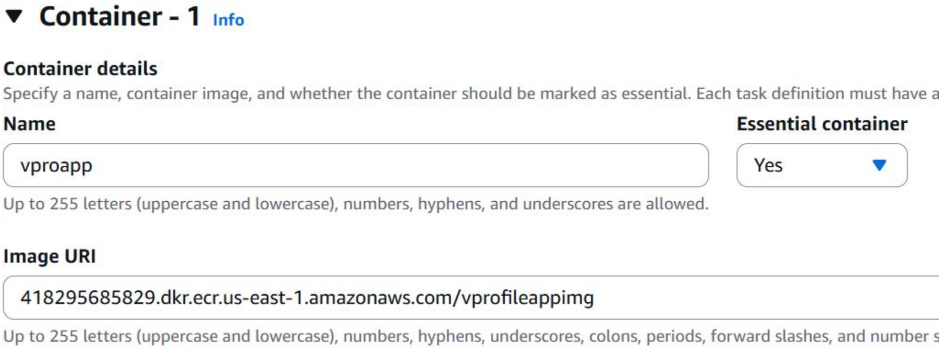
Task execution role | Info

A task execution IAM role is used by the container agent to make AWS API requ

Create new role ▼

  - Keep these 2 blank now. We'll update these later.

- In the Container section, give a name and paste the ECR registry URI in the box.



▼ Container - 1 Info

Container details

Specify a name, container image, and whether the container should be marked as essential. Each task definition must have a

Name: vproapp

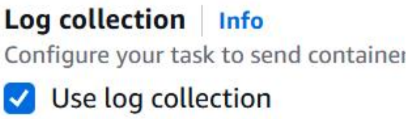
Essential container: Yes ▼

Up to 255 letters (uppercase and lowercase), numbers, hyphens, and underscores are allowed.

Image URI: 418295685829.dkr.ecr.us-east-1.amazonaws.com/vprofileappimg

Up to 255 letters (uppercase and lowercase), numbers, hyphens, underscores, colons, periods, forward slashes, and number s

- As docker is using **tomcat** and it runs on port **8080**, so give **8080** in the container port field.



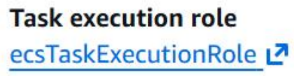
Log collection | Info

Configure your task to send containeı

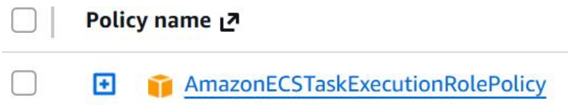
☑ Use log collection

  - Make sure this checkbox is checked. It'll upload the logs to Cloudwatch.
  - We'll update that IAM Role to access those as by default the IAM role has don't have permission to accesss the CLoudwatch logs.

- Now click on **create** button to create **task definition**.

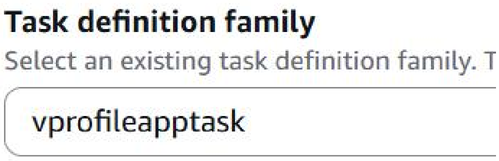
  - Now we need to provide access that IAM Role.
    - Open that **Task definition** and you'll find one link of the role.
    - 

      Task execution role

      ecsTaskExecutionRole
    - Click on this (or you can go to the IAM page and click on Role).
    - 

      □ | Policy name

      □ ⊞ AmazonECSTaskExecutionRolePolicy
    - You can see only one policy is attached to it.
    - Add the permission **CloudWatchLogsFullAccess** to that role.

- Now we'll create the **Service**
  - Go inside the cluster and create one service.
  - Now you'll get one option to choose the **Task Definition**.
    - 

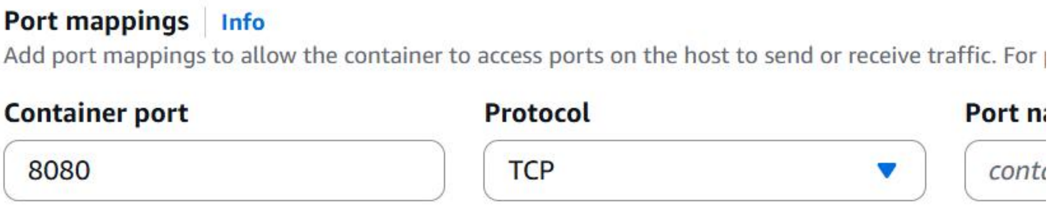
      Task definition family

      Select an existing task definition family. T

      vprofileapptask
    - Select the previously created *Task Definition*.
  - Service name: **vprofileappsvc**, Desired task: **1**,
  - For now, **un-check** the *"Use the Amazon ECS deployment circuit breaker"* under *"Deployment failure detection"* section.
  - Under the networking section, create one *security group*. This will be used by the **Load Balancer** and **Instances (which will run containers)**.
    - I added **80** and **8080** for all Ip.
  - Now under the **Load Balancer** section:
    - Select the application load balancer and give one name to that.
    - Under the listener, give the frontend port as **80**. means ELB will listen on port **80** and it'll forward to port **8080**.
    - Under **Target Group** section, give one name to the target group. Target group port: **8080**.

➢ Ports and Security groups during ECS configuration (Very Very Important) -----------

- First you create the **cluster** which is simple only.
- Then while creating **Task Definitions** :
  - Here we need to give the **Port Mapping** under the **Container** section.

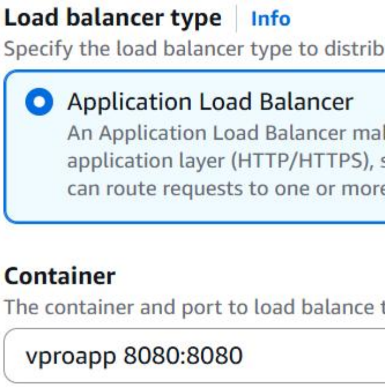


Port mappings | Info

Add port mappings to allow the container to access ports on the host to send or receive traffic. For

Container port	Protocol	Port na
8080	TCP ▼	conta

  - This port is for the container i.e. in our case tomcat will be running which listens to the port **8080** so we gave that.
  - This port mapping will appear while creating the **Service** inside the cluster selecting this **Task Definition.**



Load balancer type | Info

Specify the load balancer type to distrib

● Application Load Balancer
An Application Load Balancer mal application layer (HTTP/HTTPS), s can route requests to one or more

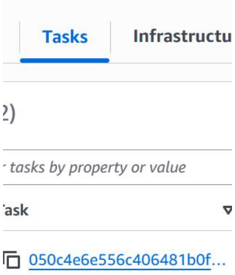
Container

The container and port to load balance t

vproapp 8080:8080

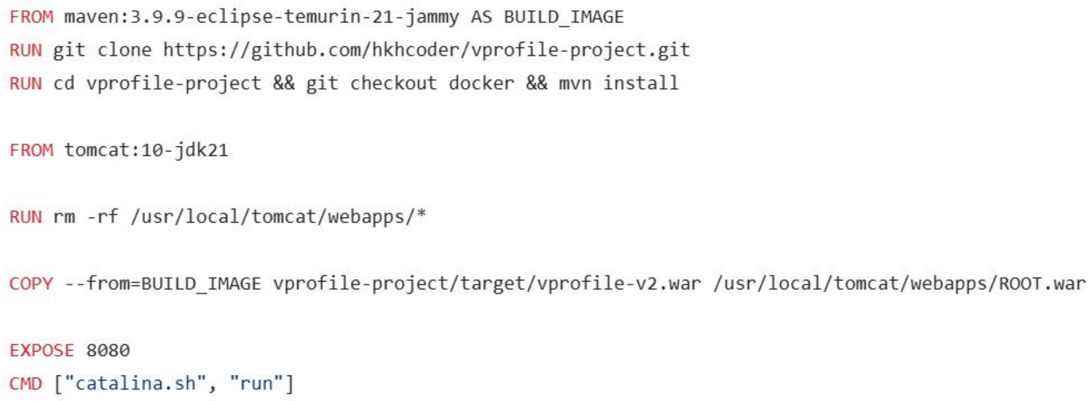
    - So here **8080:8080** mapping appeared as we gave **8080** in the Task Definition **Port Mapping.**
- Now while creating **Service** (**VERY IMPORTANT**):
  - In the beginning we need to select the **Task Definition** to let the service know about the **port mapping**, info about the **docker registry**, … etc etc.
  - Under the section **Networking**:
    - You can create or select some already created security groups.
    - Whatever the security groups will be selected or created, all of those will be attached to both **Instances** (where the docker containers will be running) and **ALB** (Application Load Balancer).
    - So, in here we need to give both the ports i.e. **80** (for ALB; Because Clients will send request to ALB with port **80**) and **8080** (for Instances; Because ALB will send traffic to Instances with port **8080**)
  - After that under the section **Load Balancing** :

    - You’ll have 2 sub-sections here; **Listener** and **Target group**.
    - Just like normal **Application Load Balancer** in EC2 service, here also this **Listener** means the port which will be accepted by **ALB**. Means this port is for **Client to ALB** connection.
    - In **Target Group** section, the port that you’ll give, will be used for **ALB to Instances** connection.
    - NOTE: This listeners and Target groups doesn’t create any security group. **Security groups** are for **IP** filtering and **Listeners** are for **Port** filtering.

- 

(The instances can be seen in Tasks tab of Cluster)
- If you want to follow the security standard i.e. **keep the security groups of *ALB* and *Instances* separately** then follow the below instructions:
    - Create 2 security groups. Lets **sg-elb** and **sg-instance**.
    - **sg-elb** inbound rule: **80** for all **IP**
    - **sg-instance** inbound rule: **8080** for **sg-elb**
    - Now, create one **Load Balancer** and attach that **sg-elb** to it.
    - Now while creating **ECS Service**, during selecting the **security group** under the section **Networking**, just select the **sg-instance**.
    - And under the **Load Balancing** section, select the created load balancer that we created before.
    - Now, the security group that we selected during creating Service i.e. **sg-instance** will not be attached to the **Load Balancer**. It’ll only be attached to the **Instance**.
    - Now the Instance is not exposed to all IP.
- ---

- A
- F
- D
- F
- A
- F
- A
- A
- F
- F
- F
- F
- D
- F
- D
- F
- F
- F
- F
- S



```
FROM maven:3.9.9-eclipse-temurin-21-jammy AS BUILD_IMAGE
RUN git clone https://github.com/hkhcoder/vprofile-project.git
RUN cd vprofile-project && git checkout docker && mvn install

FROM tomcat:10-jdk21

RUN rm -rf /usr/local/tomcat/webapps/*

COPY --from=BUILD_IMAGE vprofile-project/target/vprofile-v2.war /usr/local/tomcat/webapps/ROOT.war

EXPOSE 8080
CMD ["catalina.sh", "run"]
```

-
  - Dockerfile that is being used in build stage.
- F
- H
- A
- S
- F
- H

- **L**
- **F**
- **A**
- **S**
-