**TASK - IMPLEMENTATION OF DEVOPS USING JENKINS PIPELINE (MAVEN, SONARQUBE, JMETER, DOCKERHUB)**

**IN GOOGLE CLOUD PLATFORM**

**Step 1: Install Jenkins on VM Instance in GCP:**

Jenkins is a Java based software which can be installed from the Ubuntu packages. Jenkins is mainly used for Continues Integration and Continuous Deployment (CI CD).

**Prerequisites**

* A running Compute Engine.
* Installed Java 8.

Choose a VM Instance with at least 1 GB RAM

**Setup Firewall Rules**

Jenkins uses a custom port 8080 to run, so you need to create a firewall to all access to this port.

Go to **VPC Network >> Firewall rules** and click Create Firewall rules.

In **Name** enter Jenkins

In **Targets** select All instances in the network

In **Source filter** select IP ranges

In **Source IP ranges** enter 0.0.0.0/0

In **Protocols and ports** check **TCP** and enter 8080

Click **Create**.

**Install Jenkins**

To install latest version of Jenkins, add the repository key to the system and add the repository address to the sources list.

sudo wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -

sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'

Now you can update and install Jenkins.

sudo apt update

sudo apt install jenkins

**Starting Jenkins**

Once the installation is complete you can start Jenkins using the following command.

sudo systemctl start jenkins

You can also view the status of Jenkins using this command.

sudo systemctl status jenkins

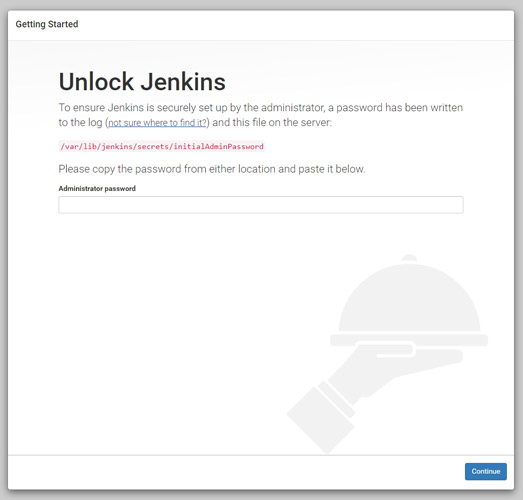
If Jenkins is started successfully you will get a response similar to this.

**Set Up Jenkins**

Once everything is done you can open your browser and enter your IP address followed by the Jenkins port 8080

The format will be like this http://instance\_external\_ip:8080

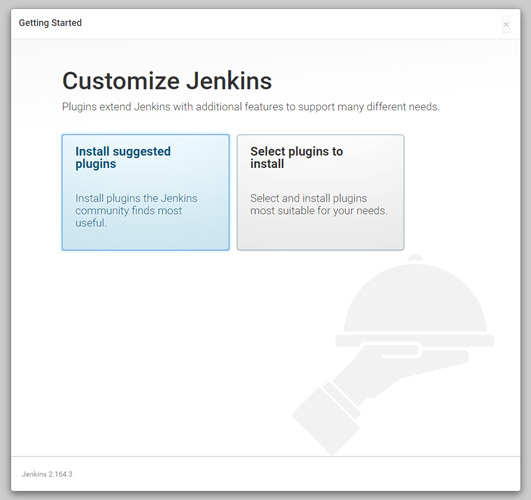
You will see the Unlock screen where you need to type the password to unlock Jenkins.



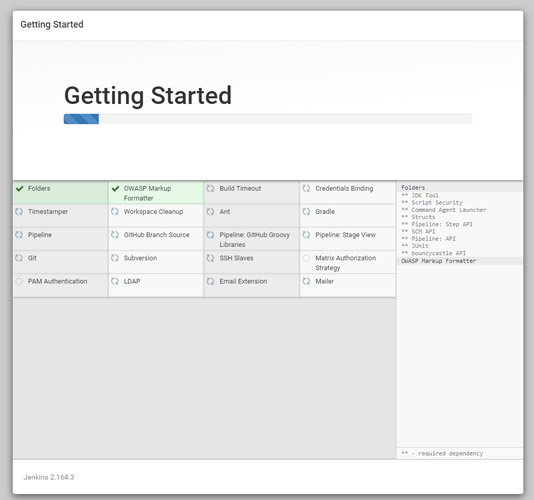
Execute the following command to get the password.

sudo cat /var/lib/jenkins/secrets/initialAdminPassword

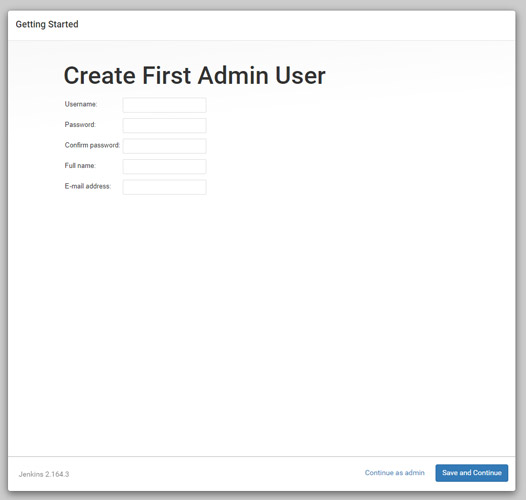
Copy the password and paste it in the **Administrator password** field to unlock and start the setup.



Click **Install suggested plugins** option to start the installation immediately.



Once the installation is complete you can create an admin user to login to the dashboard.



Finally, you will see the Instance Configuration, you can use your domain name or IP address.

Click **Save and Finish**.

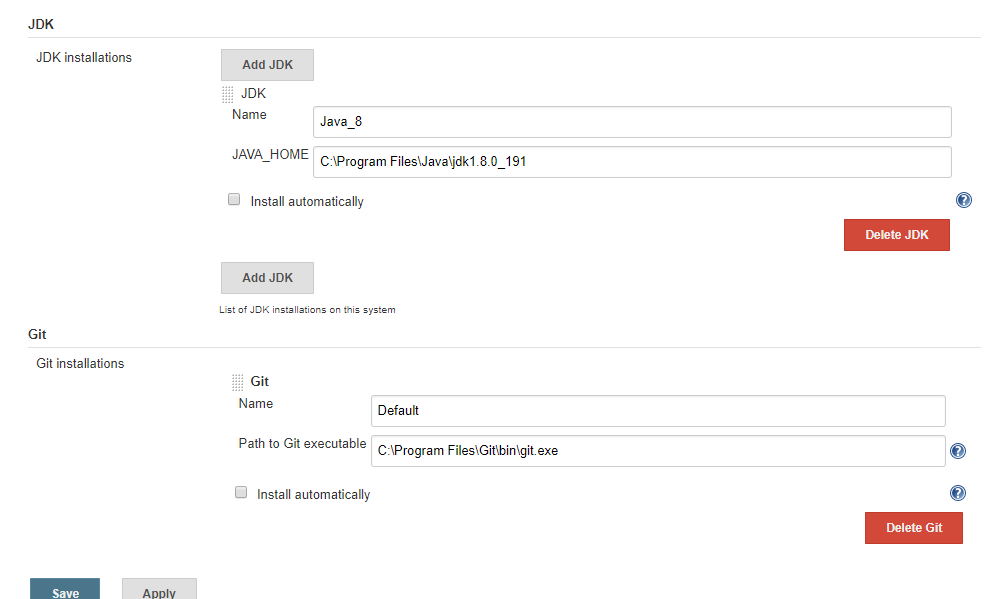
Once everything is complete click **Start using Jenkin**s to visit the main Jenkins dashboard.

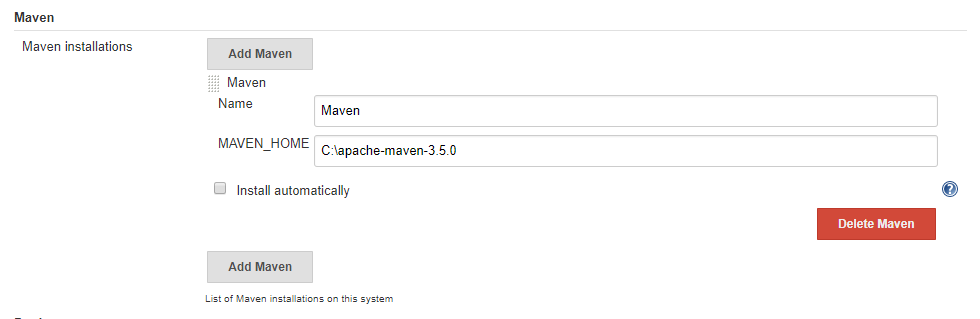
**Step 2: Integration of Github and Maven with Jenkins:**

**1.Configure Jenkins**

* Insure that **GitHub Plugin** is installed under- **Manage Jenkins** > **Manage Plugins** > **Installed** search for git. If not installed move to **Available** Tab and search for git and install it.

1. Configure *Java*, *GitHub* and *Maven* for Jenkins
2. Navigate to **Manage Jenkins** > **Global Tool Configuration**> Under **JDK** section provide *Name* and path to *JAVA\_HOME*, in same way for Git provide *Git Name* and path to Git executable, same in case of **Maven** provide *Name* and *MAVEN\_HOME* as in below images





**2.Create Job**

1. Create a new Job by clicking **New Item**

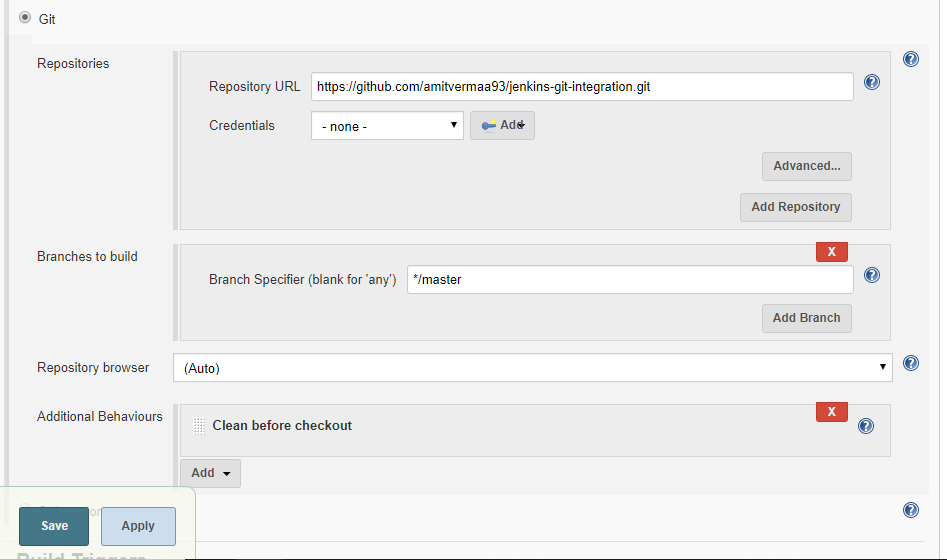
2. Enter your Job Name **‘Jenkins-GitHub’**and select ***Freestyle Project*** then click **OK**, You will be navigated to configure the Job.

3. In job configuration under **General** Section tick **Github** project and provide your *project url*from Github- **https://github.com/AbithaValli/quiz.git**this will provide you link to GitHub from Job dashboard and it is optional

4. Under **Source Code Management** section click on **Git** radio button and provide Repository URL- [**https://github.com/AbithaValli/quiz.git**](https://github.com/AbithaValli/quiz.git)

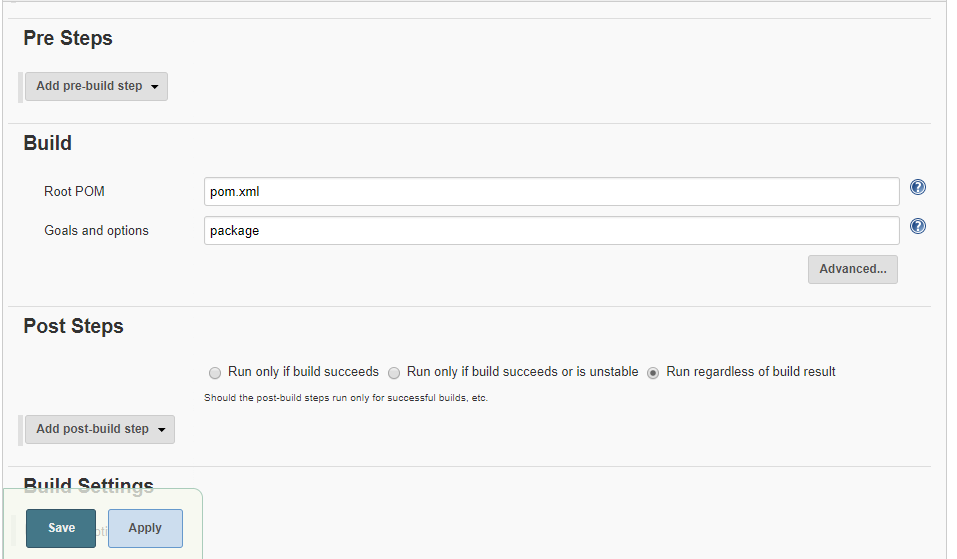
5. Select *branch* to build if master- ***‘\*/main’*** if development- ***‘\*/development’***

6. You can add *behavior* by selecting value from drop down what to perform in my case I want to clean space before code checkout so I have selected ***Clean before chekout*** (you can configure according to your need this is optional)



7. Navigate to **Build Trigger**- If you want to build your project on *specific time interval* you can configure it under **Build Trigger** > tick **Poll SCM** and in text field ***\* \* \* \* \**** (five star separated by space for every minute, see bottom of this article for convention). If you will not set up Poll you will have to manually build the Job.

8. Navigate to **Build** Section > provide path to***pom.xml***(for maven project) and *Goals and Options* **(**you can provide relative or absolute path to pom.xml, ***Relative*** : from project checkout directory as in below image, **Absolute** : from base directory like ***https://github.com/AbithaValli/quiz.git/pom.xml***and for goals you can define as you have set in your ***pom.xml***like **clean**, **package**, etc**)**



Navigate to that project. You will have to click on **Build Now** if you have not configured *Poll SCM* else it will automatically trigger every minute (as we have set \* \* \* \* \*). Your project will build successfully.

By clicking on *blue circle* you will be able to see logs.

By Clicking **#1** (#<Build Number>) you will be navigation to below window showing various project related information.

**3.Creating a pipeline job in Jenkins:**

## **Create your Pipeline project in Jenkins**

1. Go back to Jenkins, log in again if necessary and click **create new jobs** under **Welcome to Jenkins!**  
   **Note:** If you don’t see this, click **New Item** at the top left.
2. In the **Enter an item name** field, specify the name for your new Pipeline project (e.g. simple-java-maven-app).
3. Scroll down and click **Pipeline**, then click **OK** at the end of the page.
4. ( Optional ) On the next page, specify a brief description for your Pipeline in the **Description** field (e.g. An entry-level Pipeline demonstrating how to use Jenkins to build a simple Java application with Maven.)
5. Click the **Pipeline** tab at the top of the page to scroll down to the **Pipeline** section.
6. From the **Definition** field, choose the **Pipeline script from SCM** option. This option instructs Jenkins to obtain your Pipeline from Source Control Management (SCM), which will be your locally cloned Git repository.
7. From the **SCM** field, choose **Git**.
8. In the **Repository URL** field, specify the directory path of your locally cloned repository, which is from your user account/home directory on your host machine, mapped to the /home directory of the Jenkins container - i.e. - /home/Documents/GitHub/simple-java-maven-app
9. Click **Save** to save your new Pipeline project. You’re now ready to begin creating your Jenkinsfile, which you’ll be checking into your locally cloned Git repository.

# **4.Create your initial Pipeline as a Jenkinsfile**

You’re now ready to create your Pipeline that will automate building your Java application with Maven in Jenkins. Your Pipeline will be created as a Jenkinsfile, which will be committed to your locally cloned Git repository (simple-java-maven-app).

This is the foundation of “Pipeline-as-Code”, which treats the continuous delivery pipeline as a part of the application to be versioned and reviewed like any other code.

**Task 1:**

Create an initial Pipeline to download a Maven Docker image and run it as a Docker container (which will build your simple Java application). Also add a “Build” stage to the Pipeline that begins orchestrating this whole process.

Using any text editor , create and save new text file with the name Jenkinsfile at the root of your local simple-java-maven-app Git repository.

Copy the following Declarative Pipeline code and paste it into your empty Jenkinsfile:

|  |
| --- |
| pipeline { |
|  | environment { |
|  | registry = "ishu1108/docker-jenkins" |
|  | registryCredential = 'dockerhub' |
|  | dockerImage = '' |
|  | } |
|  | agent any |
|  | tools { |
|  | maven 'Maven-3.6.3' |
|  | jdk 'jdk8' |
|  | } |
|  | stages { |
|  | stage ('Initialize') { |
|  | steps { |
|  | sh ''' |
|  | echo "PATH = ${PATH}" |
|  | echo "M2\_HOME = ${M2\_HOME}" |
|  | ''' |
|  | } |
|  | } |
|  |  |
|  | stage ('Build-maven') { |
|  | steps { |
|  | echo 'This is a minimal pipeline.' |
|  | } |
|  | } |

**Step 5: Integrating SonarQube and Jenkins:**

**SonarQube Installation**SonarQube is used to analyze the source code of our project.

1. Download SonarQube from [SonarQube Downloads](https://www.sonarqube.org/downloads/).
2. Unzip the file in a directory which can be accessed without root permissions. Because SonarQube also contains the ElasticSearch which will be in SonarQube directory.So, basically, when you start SonarQube instance with root permissions it will generate an error in the start-up of ElasticSearch.
3. Before starting the server you have to set database connection with sonar. And you can find it in sonar.properties in **sonarqube/conf**
4. sonar.jdbc.url
5. sonar.jdbc.username

sonar.jdbc.password

1. SonarQuber server will start on port 9000. You can alter it in **sonar.properties**file

sonar.web.port=9000

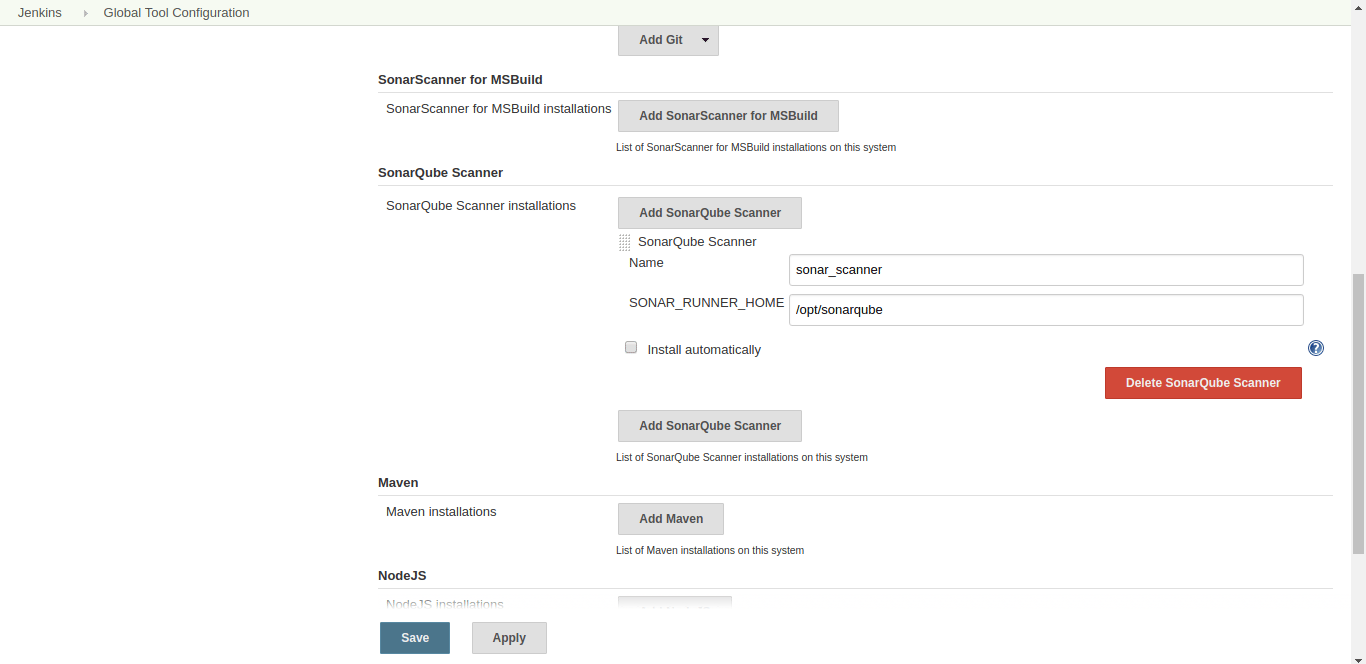
1. To start SonarQube Server from its directory **/bin/[OS directory]**

sh sonar.sh console

**Integration of SonarQube in Jenkins**  
For the integration of SonarQube in Jenkins, you have performed the following steps.

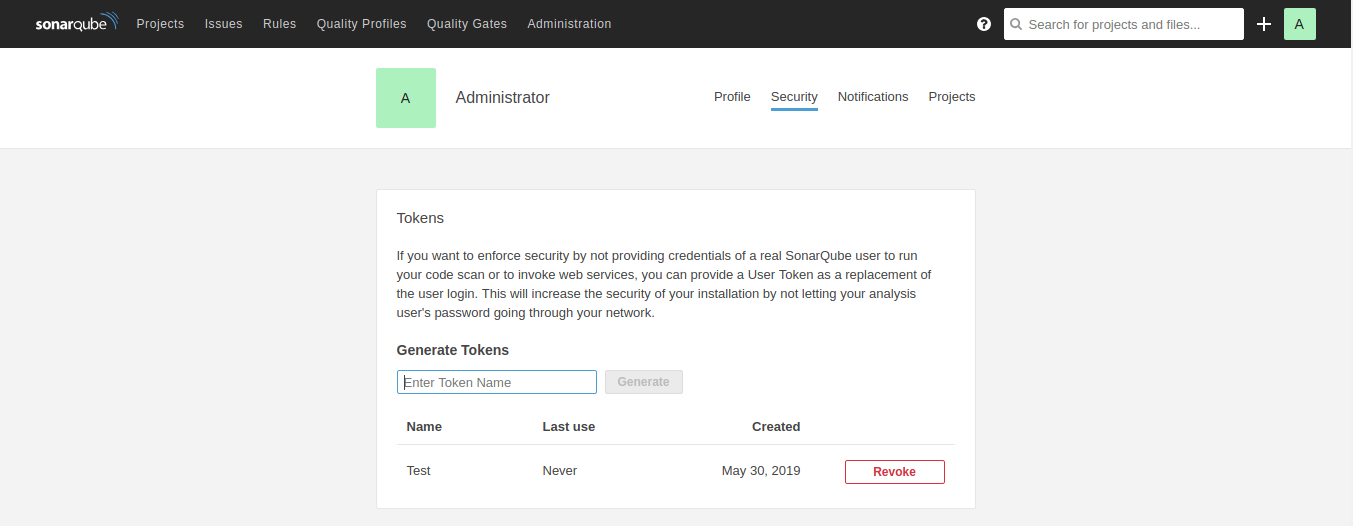
1. Login into Jenkins and install **SonarQube scanner** plugin  
   Go to **Manage Jenkins** –> **Manage Plugins** > **Available** –> **SonarQube scanner**And also add credentials plugins to store your credentials in Jenkins
2. Configure SonarQube home path  
   Go to **Manage Jenkins** –> **Global Tool Configuration** –> **SonarQube Scanner**

Name : **sonar\_scanner**

SONAR\_RUNNER\_HOME : **/opt/sonarqube**(Your directory path of SonarQube)  


Now, Configure SonarQube server in Jenkins

For integration, you need SonarQube Server authentication token in Jenkins

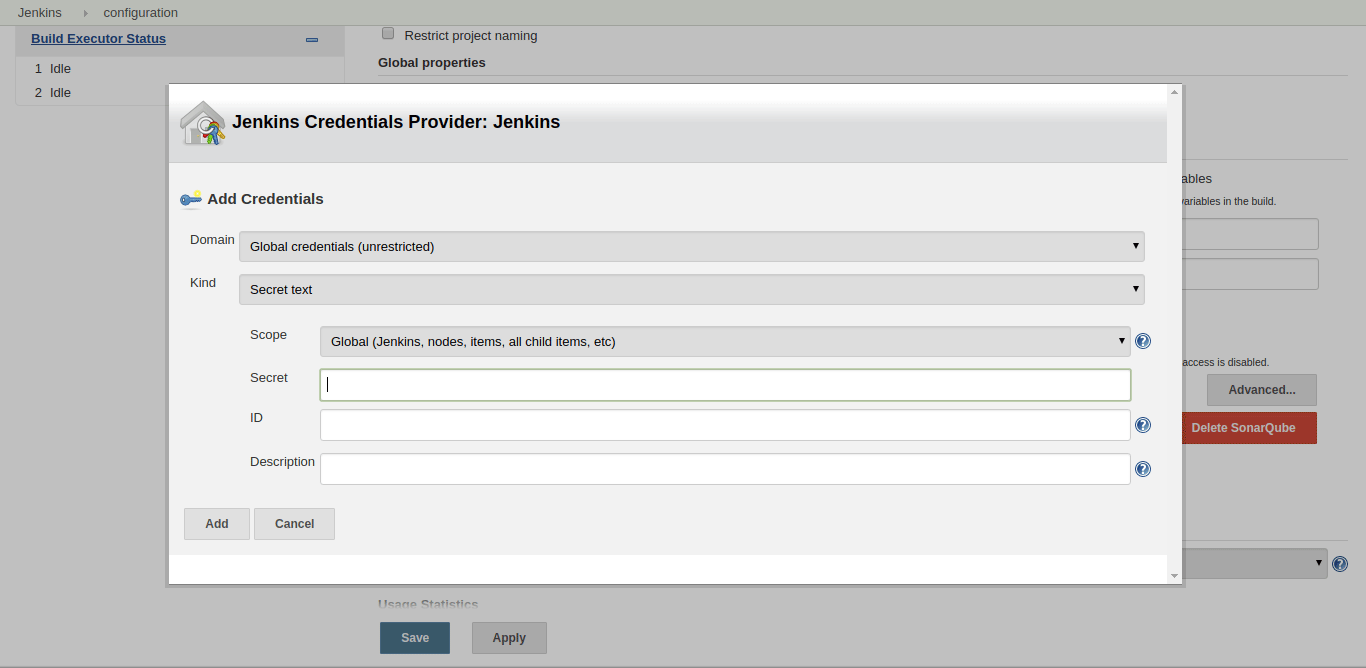
i. Log in into your SonarQube Server and find following under user bar  
Go to **My Account** –> **Security** –> **Generate Token**  


Go to **Manage Jenkins** –> **Configure Systems** –> **SonarQube Servers**

Name : **SonarQube**  
Server URL : Not Required is same as default

Server authentication token :

i. Add server authentication token as following



ii. Select it as a server authentication token

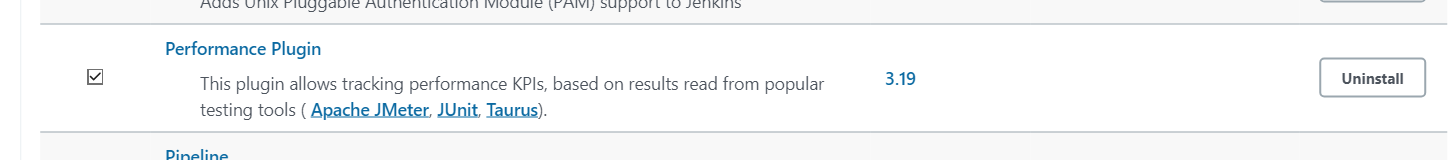
Save it. Now, your SonarQube integration is completed with Jenkins. Create a job (Follow [Jenkins – Continuous Integration System](https://aspiresoftware.in/blog/jenkins-continuous-integration-system/)) to test SonarQube and to generate a report of your project.

**Step 6: Using Jmeter in Jenkins:**

**1.Installing Plugins:**

**First step is installing the performance plugins.**

1. Go to Manage Jenkins and select Manage Plugins.
2. Install Performance Plugin from the available plugins if you don’t have it already installed.



**Installing Jmeter on your device.**

1. You can download jmeter directly from https://jmeter.apache.org/download\_jmeter.cgi by clicking either the zip file or tgz file in the binaries.

Since we are using Debian, we installed it using the following commands from the terminal,

wget [https://mirrors.estointernet.in/apache//jmeter/binaries/apache-jmeter-5.4.1.zip](https://mirrors.estointernet.in/apache/jmeter/binaries/apache-jmeter-5.4.1.zip)

unzip [https://mirrors.estointernet.in/apache//jmeter/binaries/apache-jmeter-5.4.1.zip](https://mirrors.estointernet.in/apache/jmeter/binaries/apache-jmeter-5.4.1.zip)

**Modifying user. config file**

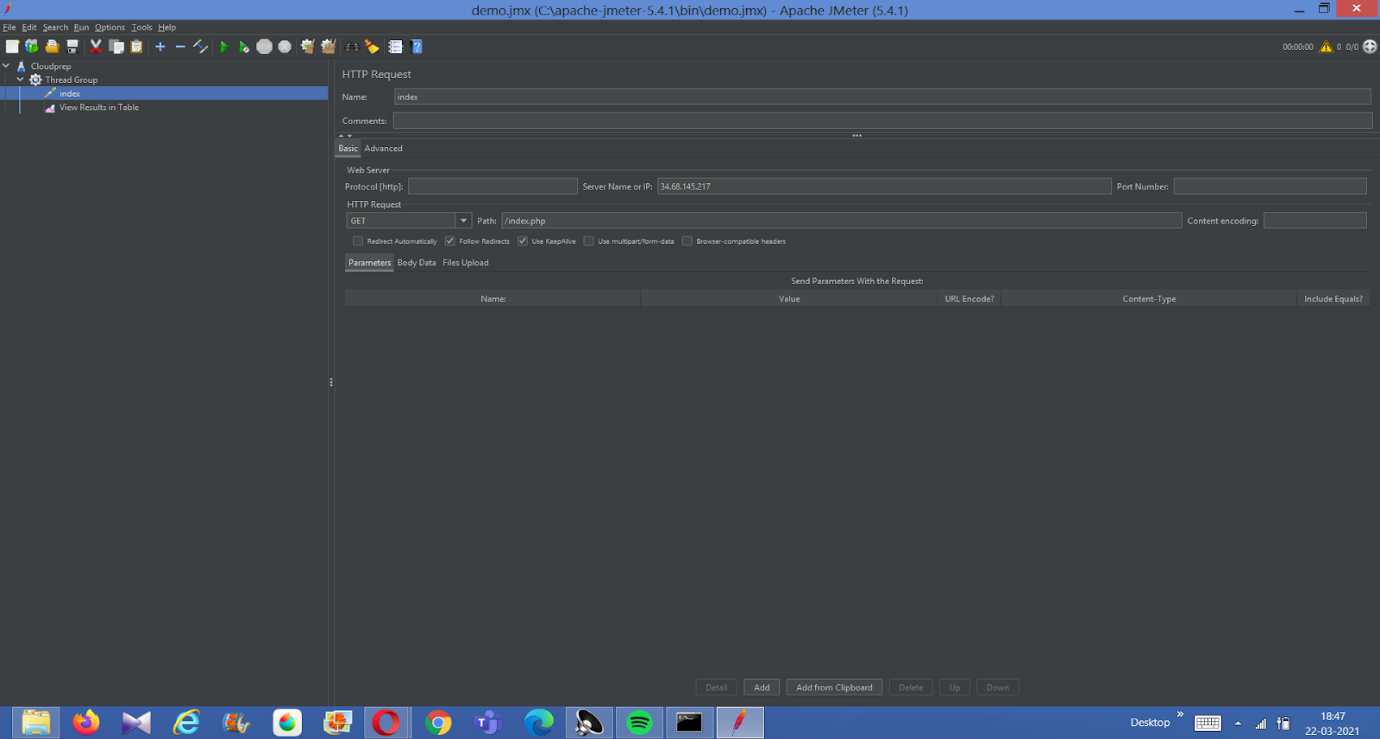
a. Go to the bin directory and open user. config file.

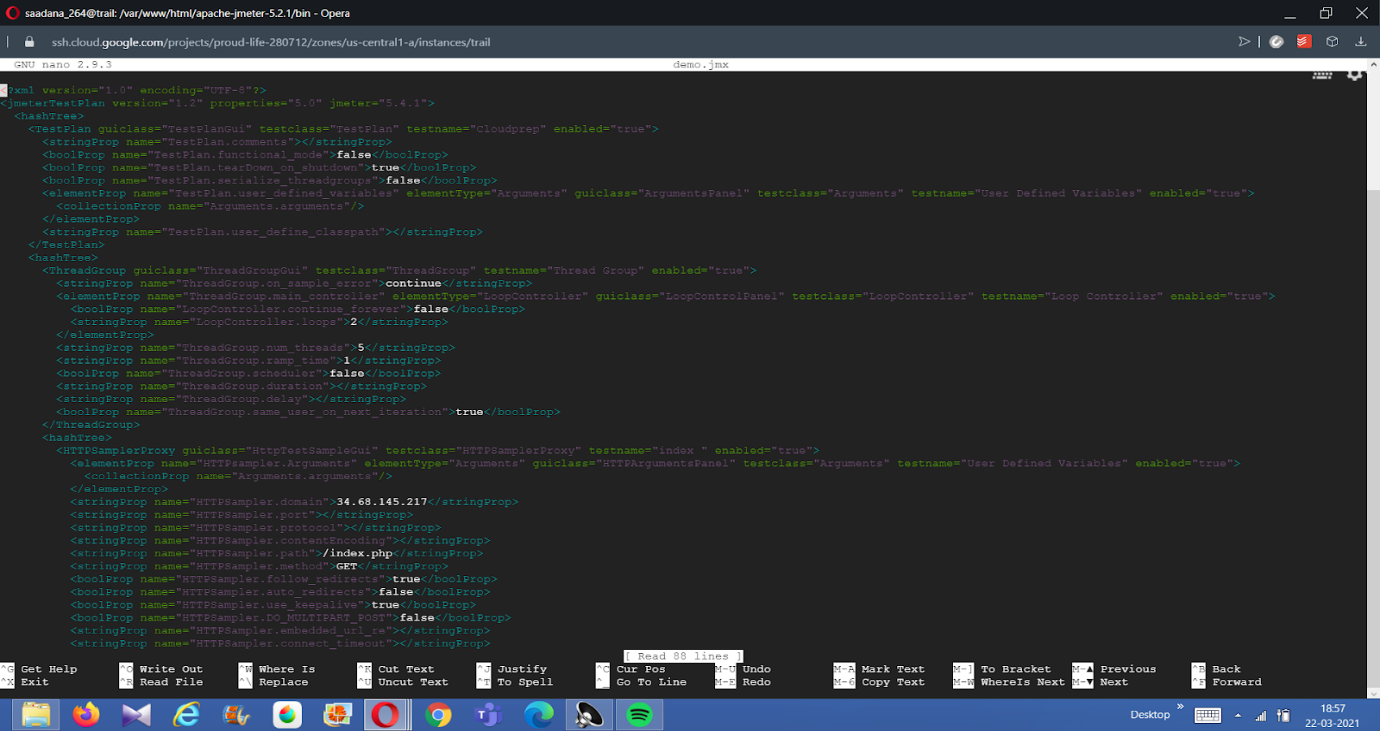
b. Add the line ‘jmeter.save.saveservice.output\_format =x ml’ at last

**Creating jmx file**

1. Open jemeter.bat and create a thread group for adding pages you want to do a performance test on.

Or you can directly create a jmx file and write the code inside if you prefer that.





Run the command to execute performance testing

**Step 7: Integrating Docker Hub and Jenkins:**

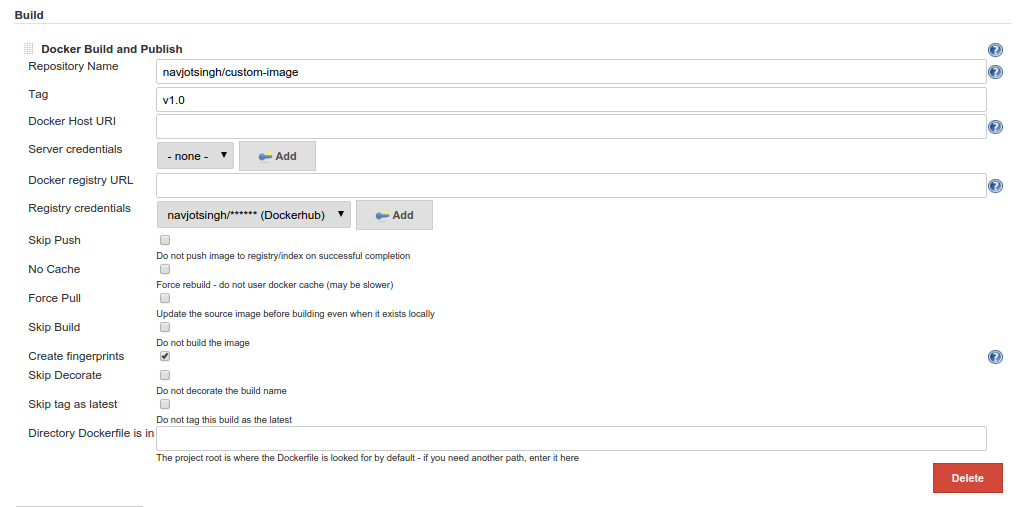
**CloudBees Docker Build and Publish plugin**:   
It enables us to build images from the Docker file present on the server and publishing them on the Docker Hub.

**CloudBees Docker Hub Notification**:   
It enables us to trigger one or more Jenkins job by making use of Docker Hub’s web hook, thus creating a continuous delivery pipeline. Whenever a new image is pushed, the configured Jenkins job will receive notification as web hook and triggers the job.

**Scenario:** One click automation of a process of building image from Dockerfile, pushing it on DockerHub and immediately triggering a Jenkins deployment job based on the pushed docker image.

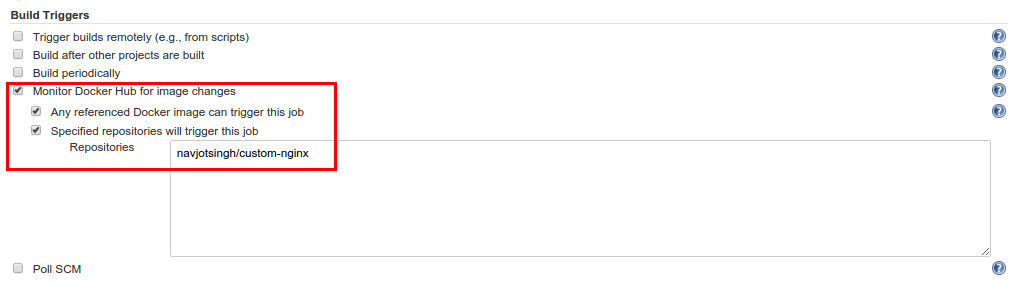
**Installing Plugins:**  
Install the above two plugins using Jenkins’ “**Plugin Manager**”.

**Creating and configuring Jenkins job to build images from Dockerfile:**  
Create a new Jenkins job (say “*Build Docker Image*“) which will use *CloudBees Docker Build and Publish plugin* to build images from Dockerfile and push it on DockerHub.  
Configure this job as follows under the build section:



Various fields in the above image are explained below:

* + - **Repository Name**: As shown we have passed “navjotsingh/custom-image” as the repository name.
    - **Tag**: We are passing “v1.0” as tag or version. We can make this build parametrized and pass custom repository name and tag as two parameters to this job.
    - **Docker Host URI**: Our Docker host is on the same machine where Jenkins is installed, so we are using default Docker ‘s default URI by keeping this field empty.
    - **Server credentials**: Provide user’s credentials who has permission to run Docker commands and have permission to use Docker’s socket “/var/run/docker.sock”. I have provided permission to Jenkins’ user.
    - **Docker registry URL**: We are using public DockerHub registry so we have left “Docker registry URL” field empty.
    - **Registry credentials**: Click “Add” to add the DockerHub account credentials.
    - As we can see there are various options provided by this plugin wherein we can choose to **skip push, no cache, force pull and skip build**.
    - This plugin creates **fingerprints** after building image and are managed by Docker Commons Plugin (installed with this plugin).
    - Decorating the build name means builds will be decorated with the repository name and tag name. We can skip it by checking “**Skip Decorate**” check box.Jenkins6
    - **Skip tag as latest**: If it is unchecked, it will create additional tag “latest” of the currently build image and push it on Docker Hub as can be seen in the above image.
    - **Docker Directory is in**: The directory which contain the Dockerfile can be specified in this field otherwise it will try to fetch the Dockerfile from the Job’s workspace.

* **Modifying existing Deployment Job to be triggered immediately after building image:**  
  Now we will modify the existing deployment job which we need to be executed whenever that Docker image is updated. We will achieve it by using **CloudBees Docker Hub Notification plugin**.  
  Configure the deployment job under “Build Triggers” as follows:  
  Select “**Monitor Docker Hub image changes**”, under this select “**Any referenced Docker image can trigger this job**”. We can make this job to be triggered on specified repositories by selecting “**Specified repositories will trigger this job**” and specifying repository name against “**Repositories**” field. Multiple repositories can be specified in the text field, one repository per line.  
  Now, we have configured our deployment job to be triggered if any new image gets available in navjotsingh/custom-image repository.

# **Jenkins Pipeline to Create Docker Image and Push to Dockerhub**

|  |
| --- |
| pipeline { |
|  | environment { |
|  | registry = "ishu1108/docker-jenkins" |
|  | registryCredential = 'dockerhub' |
|  | dockerImage = '' |
|  | } |
|  | agent any |
|  | tools { |
|  | maven 'Maven-3.6.3' |
|  | jdk 'jdk8' |
|  | } |
|  | stages { |
|  | stage ('Initialize') { |
|  | steps { |
|  | sh ''' |
|  | echo "PATH = ${PATH}" |
|  | echo "M2\_HOME = ${M2\_HOME}" |
|  | ''' |
|  | } |
|  | } |
|  |  |
|  | stage ('Build-maven') { |
|  | steps { |
|  | echo 'This is a minimal pipeline.' |
|  | } |
|  | } |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | stage('Build') { |
|  | steps { |
|  | echo 'Building...' |
|  | echo "Running ${env.BUILD\_ID} ${env.BUILD\_DISPLAY\_NAME} on ${env.NODE\_NAME} and JOB ${env.JOB\_NAME}" |
|  | } |
|  | } |
|  | stage('Building our image') { |
|  | steps{ |
|  | script { |
|  | dockerImage = docker.build registry + ":$BUILD\_NUMBER" |
|  | } |
|  | } |
|  | } |
|  | stage('Deploy our image') { |
|  | steps{ |
|  | script { |
|  | docker.withRegistry( '', registryCredential ) { |
|  | dockerImage.push() |
|  | } |
|  | } |
|  | echo 'http://34.68.145.217/' |
|  | } |
|  | } |
|  | } |
|  |  |

