JENKINS SETUP

#!/bin/bash

sudo apt update -y

sudo apt upgrade -y

wget -O - https://packages.adoptium.net/artifactory/api/gpg/key/public | tee /etc/apt/keyrings/adoptium.asc

echo "deb [signed-by=/etc/apt/keyrings/adoptium.asc] https://packages.adoptium.net/artifactory/deb $(awk -F= '/^VERSION\_CODENAME/{print$2}' /etc/os-release) main" | tee /etc/apt/sources.list.d/adoptium.list

sudo apt update -y

sudo apt install temurin-17-jdk -y

/usr/bin/java --version

curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key | sudo tee \

/usr/share/keyrings/jenkins-keyring.asc > /dev/null

echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] \

https://pkg.jenkins.io/debian-stable binary/ | sudo tee \

/etc/apt/sources.list.d/jenkins.list > /dev/null

sudo apt-get update -y

sudo apt-get install jenkins -y

sudo systemctl start jenkins

sudo systemctl status jenkins

INSTALL SONARQUBE ON UBUNTU 22.04

Install docker on it

sudo apt update

sudo apt install docker.io -y

sudo systemctl start docker

sudo systemctl enable docker

sudo systemctl status docker

Sudo su –

docker run –d –p 9000:9000 sonarqube:lts

docker images

docker ps (check its running or not )

<public-ip:9000>

Username admin

Password admin

INSTALL NEXUS on ubuntu 22.04

Sudo su –

sudo apt update

sudo apt install docker.io -y

sudo systemctl start docker

sudo systemctl enable docker

sudo systemctl status docker

docker run –d –p 8081:8081 sonatype/nexus3

docker exec –it <container-id> bash

cd /nexus-data/

ls –l

cat admin.pasword

there you can find credentials of nexus

1. Create Nexus volume:

<!---->

Copy code

docker volume create nexus-data

2. Start Nexus container with volume:

<!---->

Copy code

docker run -d --name nexus -p 8081:8081 --mount source=nexus-data,target=/nexus-data sonatype/nexus3

3. Login to Nexus UI at [http://your-ip:8081](http://your-ip:8081/) and create a Docker hosted repo called "docker-private" 4. Stop and remove the Nexus container:

<!---->

Copy code

docker stop nexus

docker rm nexus

5. Start Nexus container with port mapping for Docker registry:

<!---->

Copy code

docker run -d --name nexus -p 8081:8081 -p 8083:8083 --mount source=nexus-data,target=/nexus-data sonatype/nexus3

6. Login at [http://your-ip:8081](http://your-ip:8081/) and edit the Docker hosted repo to use port 8083 7. Access registry at <http://your-ip:8083/v2/_catalog>

KUBERNTES SETUP

Part 1 ---------------------------------------------------

Master ------------

sudo su

hostname master

bash

clear

Node --------------

sudo su

hostname master

bash

clear

Part 2 -----------------------------------------------------

Both Master & Node -------------

sudo apt-get update && sudo apt-get upgrade -y

sudo apt-get install -y docker.io

sudo curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -

sudo tee /etc/apt/sources.list.d/kubernetes.list <<EOF

deb https://apt.kubernetes.io/ kubernetes-xenial main

EOF

sudo apt-get update

sudo apt-get install -y kubelet kubeadm kubectl

snap install kube-apiserver

Part 3 -----------------------------------------------------

Master ---------------

sudo kubeadm init --pod-network-cidr=10.244.0.0/16

mkdir -p $HOME/.kube

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown $(id -u):$(id -g) $HOME/.kube/config

kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml

Node ----------

paster the kube adm join command which is in this format: sudo kubeadm join <master-node-ip>:<master-node-port> --token <token> --discovery-token-ca-cert-hash <hash>

Part 4 ----------------------------------------------------------------

Master -----------------

kubectl get nodes

---------------------------------------------------------------------------------------------------------------

CONGRATULATIONS FOR YOUR NEW KUBERNETES CLUSTER ON UBUNTU ON EC2

INSTALL HELM ON JENKINS MACHINE

Download helm

curl -fsSL -o get\_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3

chmod 700 get\_helm.sh

./get\_helm.sh (RUN IT)

helm

helm –version

and install docker also

sudo apt install docker.io –y

sudo systemctl enable docker

sudo usermod –aG docker Ubuntu

sudo chmod 666 /var/run/docker.sock

sudo apt-get install unzip

INSTALL DATREE PLUGIN

Go to browser

Datree.io and click on docs

In the left side panel click on integrations

Click on Helm plugin in left side panel and

Paste this link in your Ubuntu 22.04 jenkins machine to install datree plugin

helm plugin install <https://github.com/datreeio/helm-datree>

**Phase II: The CICD Pipeline**

* In this phase we will do everything from integrating our Jenkins, SonarQube, Nexus Repository Manager, Kubernetes cluster, Helm, [datree.io](http://datree.io/) applications with each other to deploying our Java web application on the K8s cluster using Helm charts based on pull request triggers. Let's begin.

**Clone the git repository**

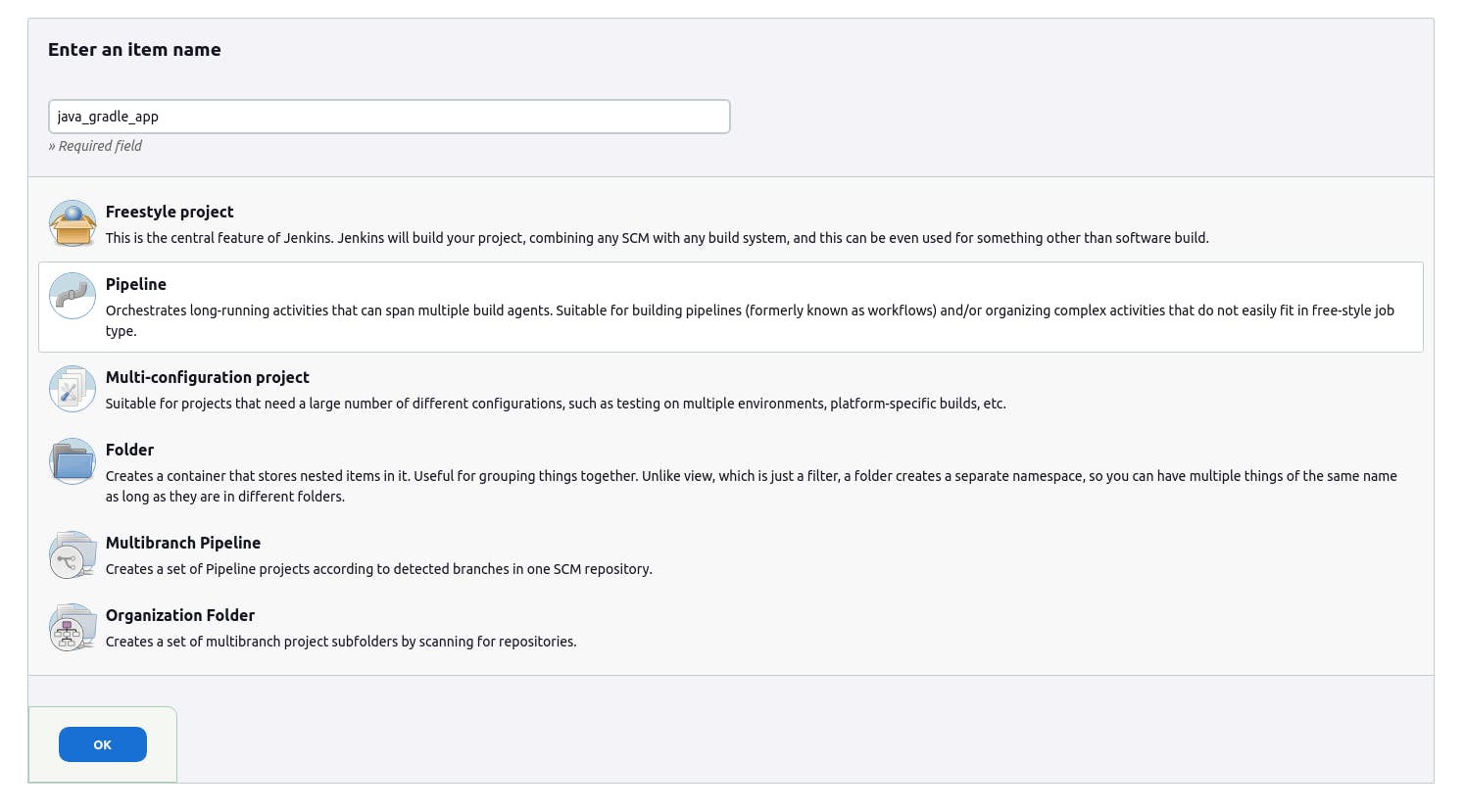
* We will be updating and pushing the changes to the repository so we need to clone the repository via ssh to do so first create a fork of the repository and then

git clone

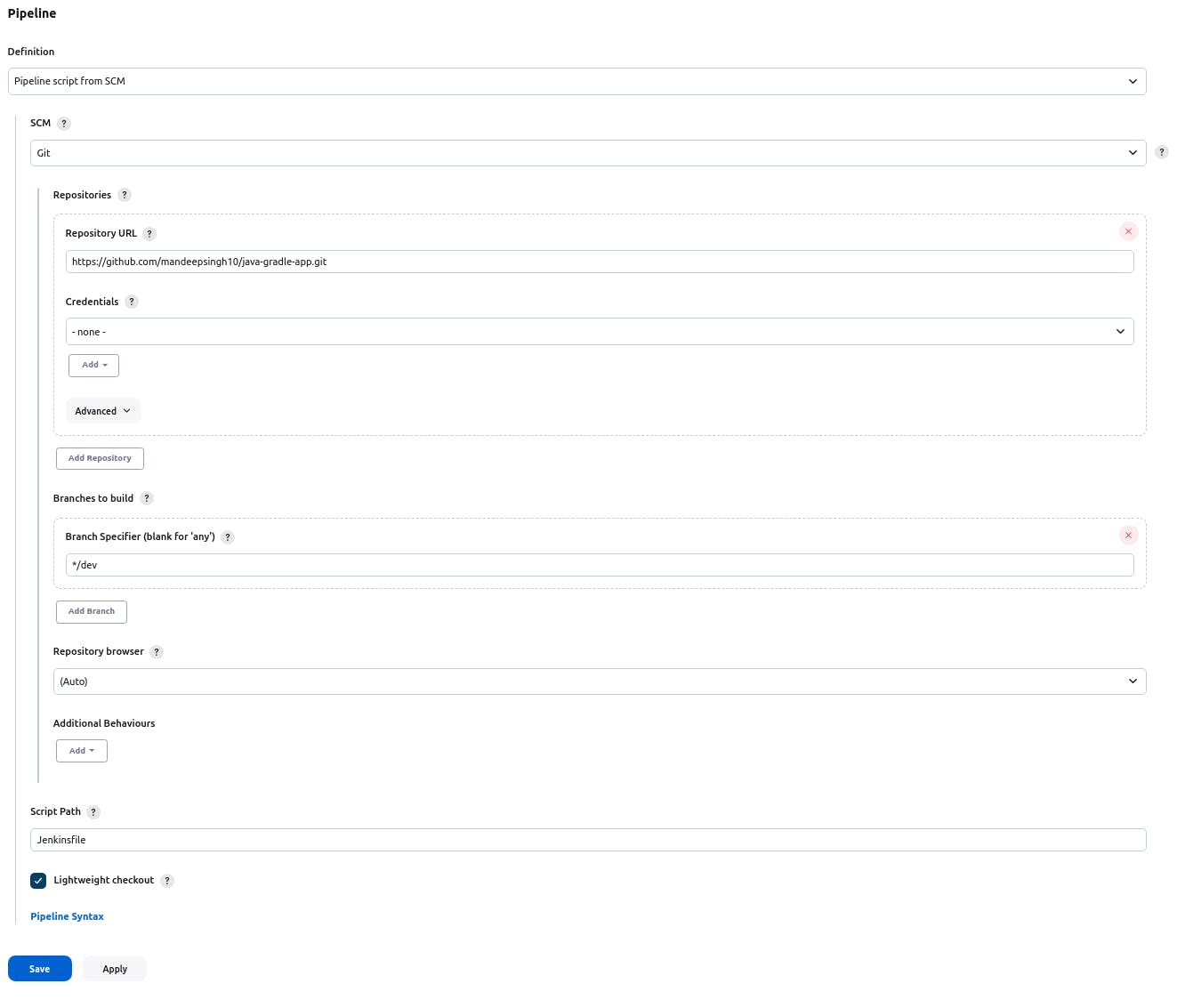
* Currently we only have main branch in the repository, so we will create a new branch called dev as it is not a good practice to modify the main branch.

**Creating a pipeline job**

* Go to the jenkins dashboard and click on New Item, select the type as Pipeline and Enter a name for the pipeline. Click on OK.



* Navigate to the Pipeline section of the job and select Pipeline script from SCM, select SCM as Git, enter the repository URL, leave the Credentials field empty as our repo is public. In the Branches to build section, select Branch Specifier as the newly created dev branch. Click on Save.



* Now, let's create a Jenkinsfile in the dev branch with a simple one stage pipeline to check if our code is getting pulled properly or not. Open your favorite code editor and create the Jenkinsfile in the root directory of the repo, make sure you are checked out to dev branch.

# pipeline{

# agent any

# stages{

# stage("Git Test"){

# steps{

# echo "Executing git connection test"

# }

# post{

# success{

# echo "git repository cloned successfully"

# }

# failure{

# echo "git clone action failed"

# }

# }

# }

# }

# post{

# success{

# echo "========pipeline executed successfully ========"

# }

# failure{

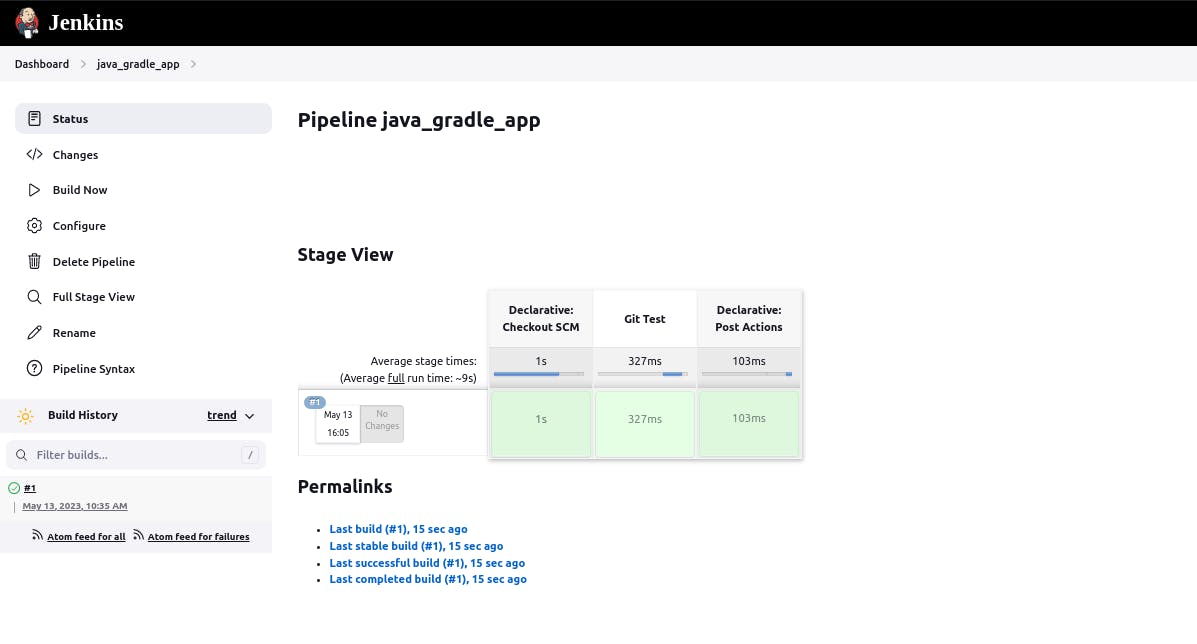
# echo "========pipeline execution failed========"

# }

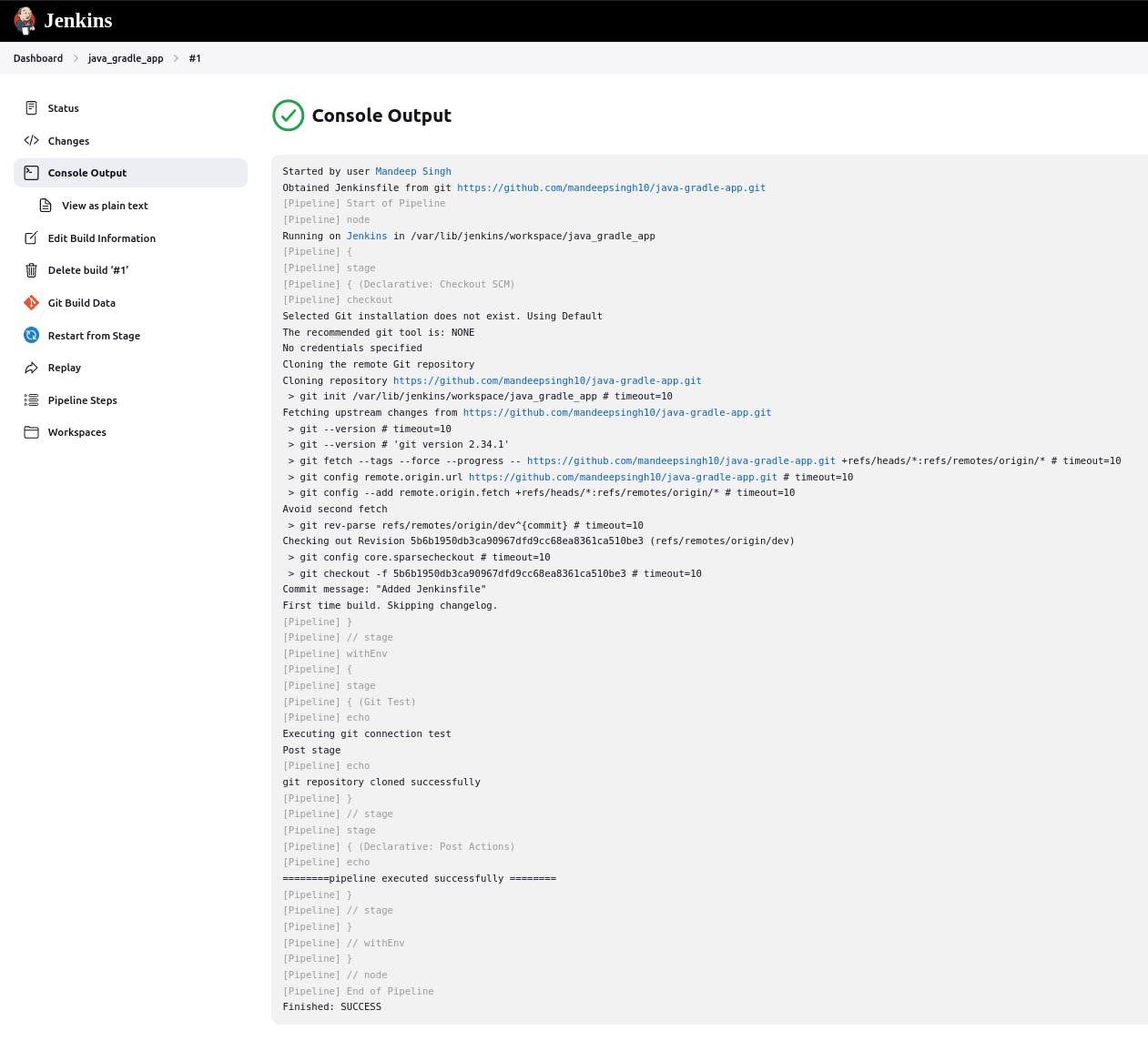
# }

# }

* Add the Jenkinsfile to dev branch and commit the changes.
* Now, let's try building our pipeline job in jenkins. Goto Jenkins dashboard and then select java-gradle-app and click on Build Now.



* Our test build was successfull. Let's check the logs, click on the #1 build and then select Console Output.



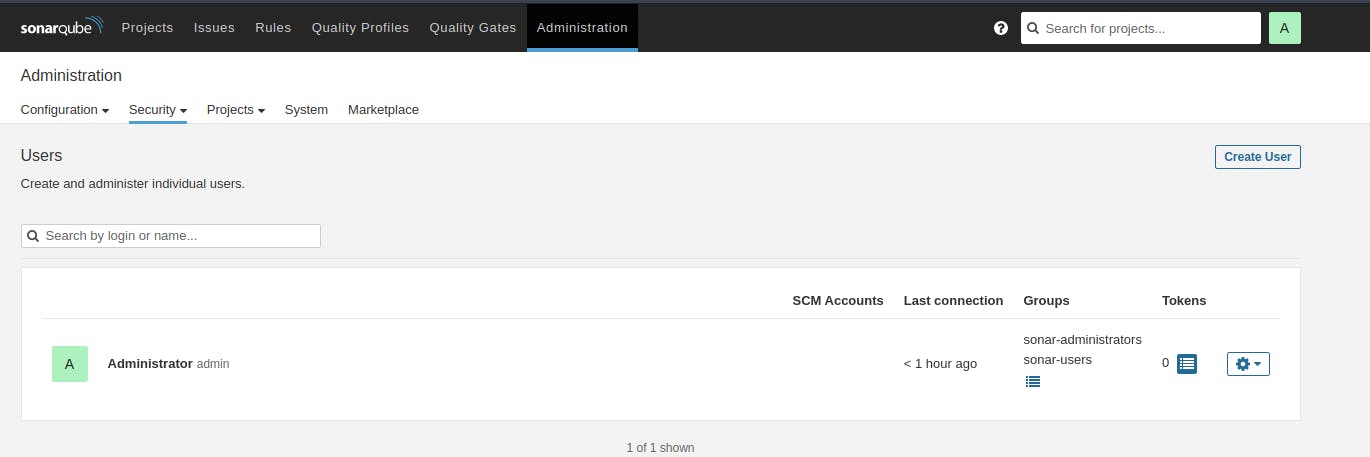
We can see the commands that were executed during the execution and some other details, this completes our test run, now let's integrate SonarQube with our Jenkins server.

**STAGE I : SonarQube Integration**

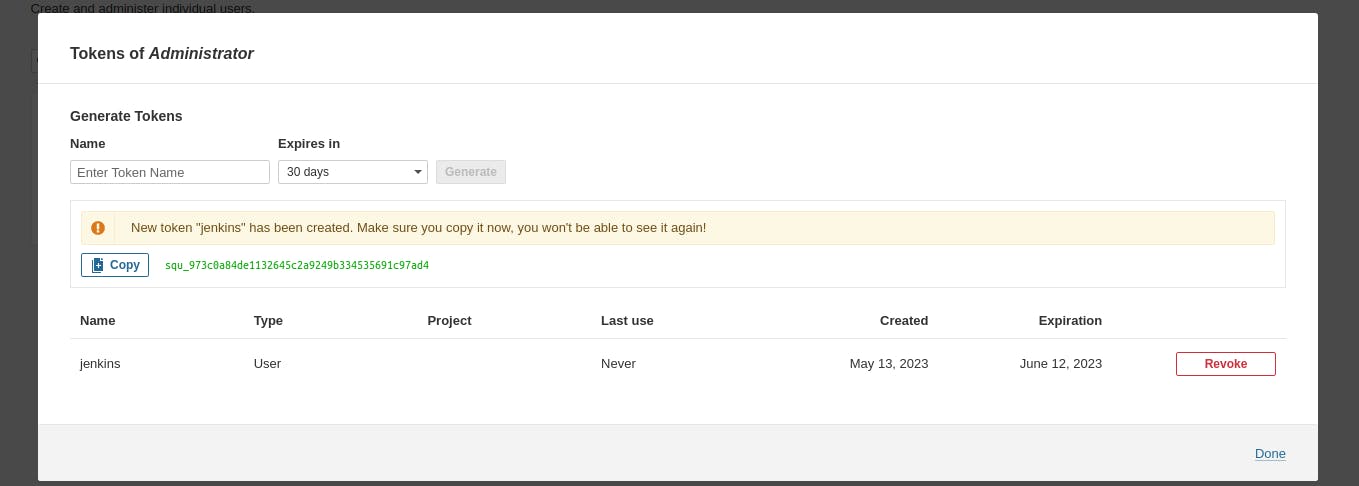
* In this step, we will integrate our SonarQube server with our Jenkins server so that we can perform Code Quality checks, code smells etc.
* To start, we'll need to install some jenkins plugins, to do so, navigate to  
  Manage Jenkins > Manage Plugins > Available and install the plugins listed below:
  1. SonarQube Scanner for Jenkins
  2. Sonar Gerrit
  3. Sonar Quality Gates
  4. SonarQube Generic Coverage
  5. Quality Gates
  6. Docker
  7. Docker Pipeline
  8. docker-build-step

Click on Install without restart, it will take some time to install the plugins.

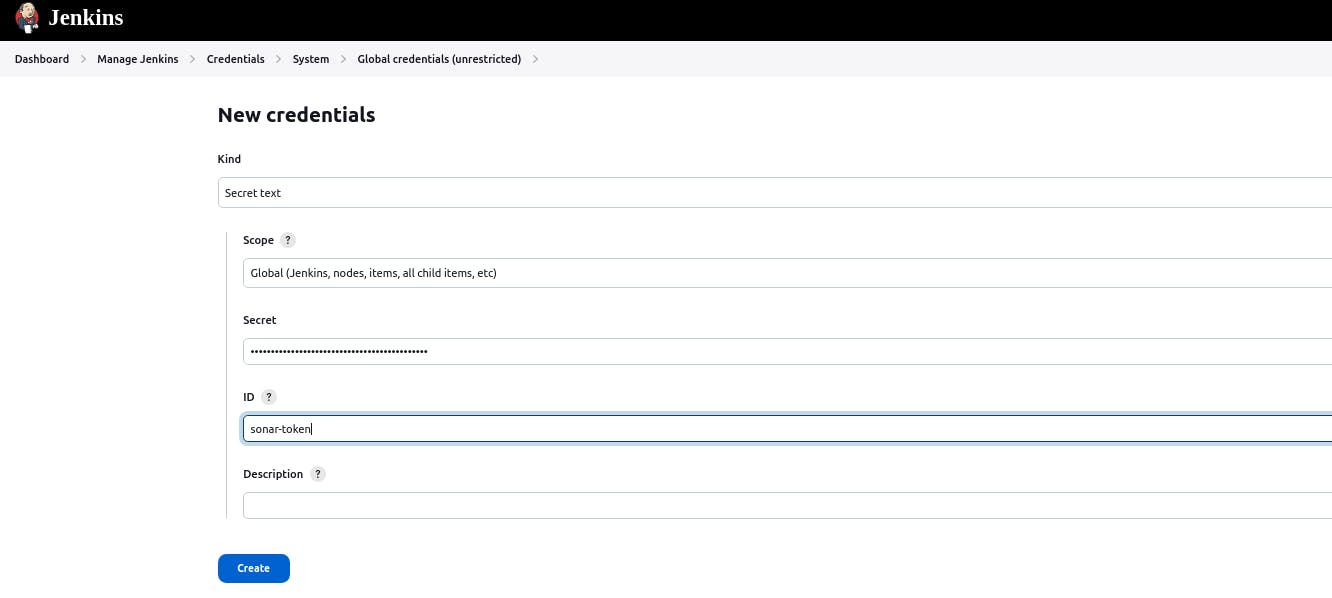
* Next, we will establish connection between SonarQube and Jenkins Server for that we will need an authentication token from SonarQube.  
  Goto SonarQube dashboard, goto Administration > Security, generate a token for administrator user by clicking on the icon below the Tokens column.



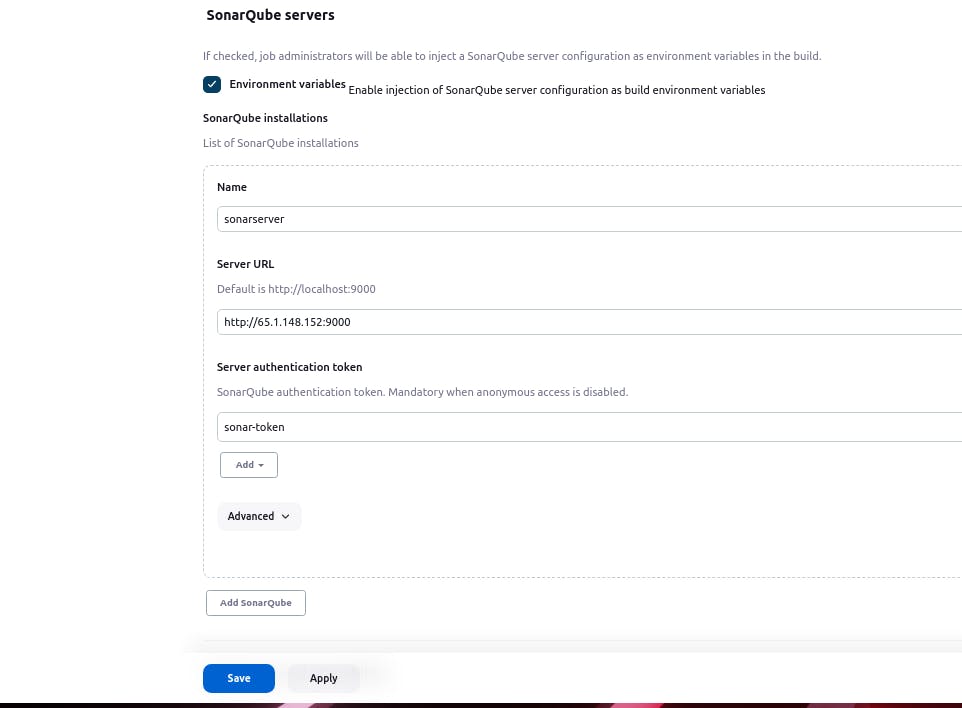
Choose name as jenkins and leave the Expires in field at the default value of 30 days and click on Generate.



* Copy this token, we will create a Jenkins secret using this token which will used to authenticate jenkins while connecting with sonarqube.  
  Goto Manage Jenkins > Security > Credentials. Click on global and then add credentials, select Kind as Secret text and ID as sonar-token



* Now, we need to configure SonarQube settings, Goto Manage Jenkins > Configure System . There will be a section SonarQube servers, we need to update the details in this section. Click on Add SonarQube, Enter the name as sonarserver, Server URL is http://IP:PORT, select Server authentication token as sonar-token. Also enable Environment variables.



* We need to use this token in our Jenkinsfile so we will need to generate a pipeline script for that, navigate to the pipeline job and click on Pipeline Syntax and generate sonarqube pipeline script from the **Snippet Generator**.  
  Select Sample Step as withSonarQubeEnv , select Server authentication token as sonar-token, click on Generate Pipeline Script.

withSonarQubeEnv {

// some block

}

This is the pipeline script generated, similarly whenever we need to generate any script we can use the **Snipper Generator**.

* Now add this pipeline script to the Jenkinsfile. Remove the previous test stages and create a new Stage Sonar Quality Check where we will perform code analysis by querying SonarQube.

COPY

pipeline{

agent any

environment{

VERSION = "{env.BUILD\_ID}"

}

stages{

stage("Sonar Quality Check"){

steps{

script{

withSonarQubeEnv(credentialsId: 'sonar-token') {

sh 'chmod +x gradlew'

sh './gradlew sonarqube --info'

}

}

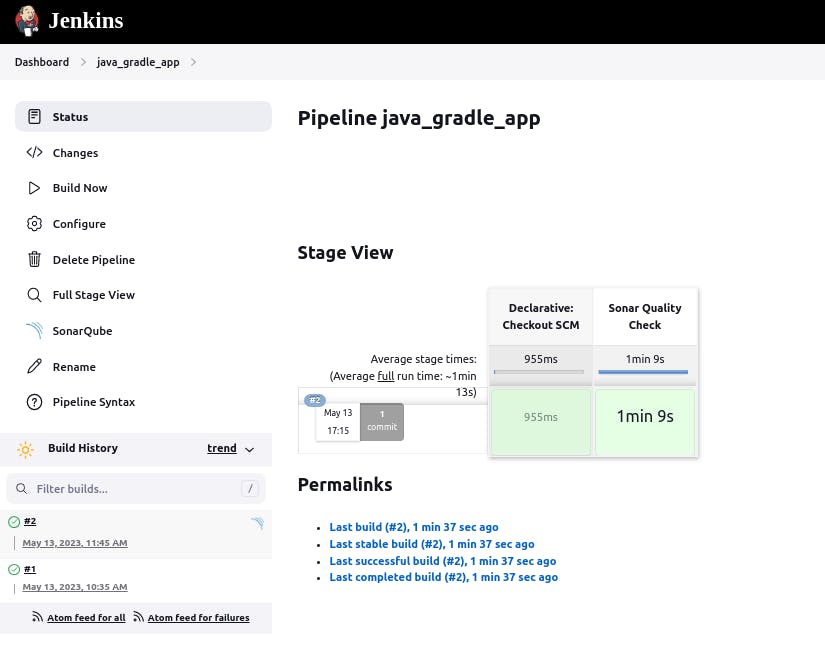
}

}

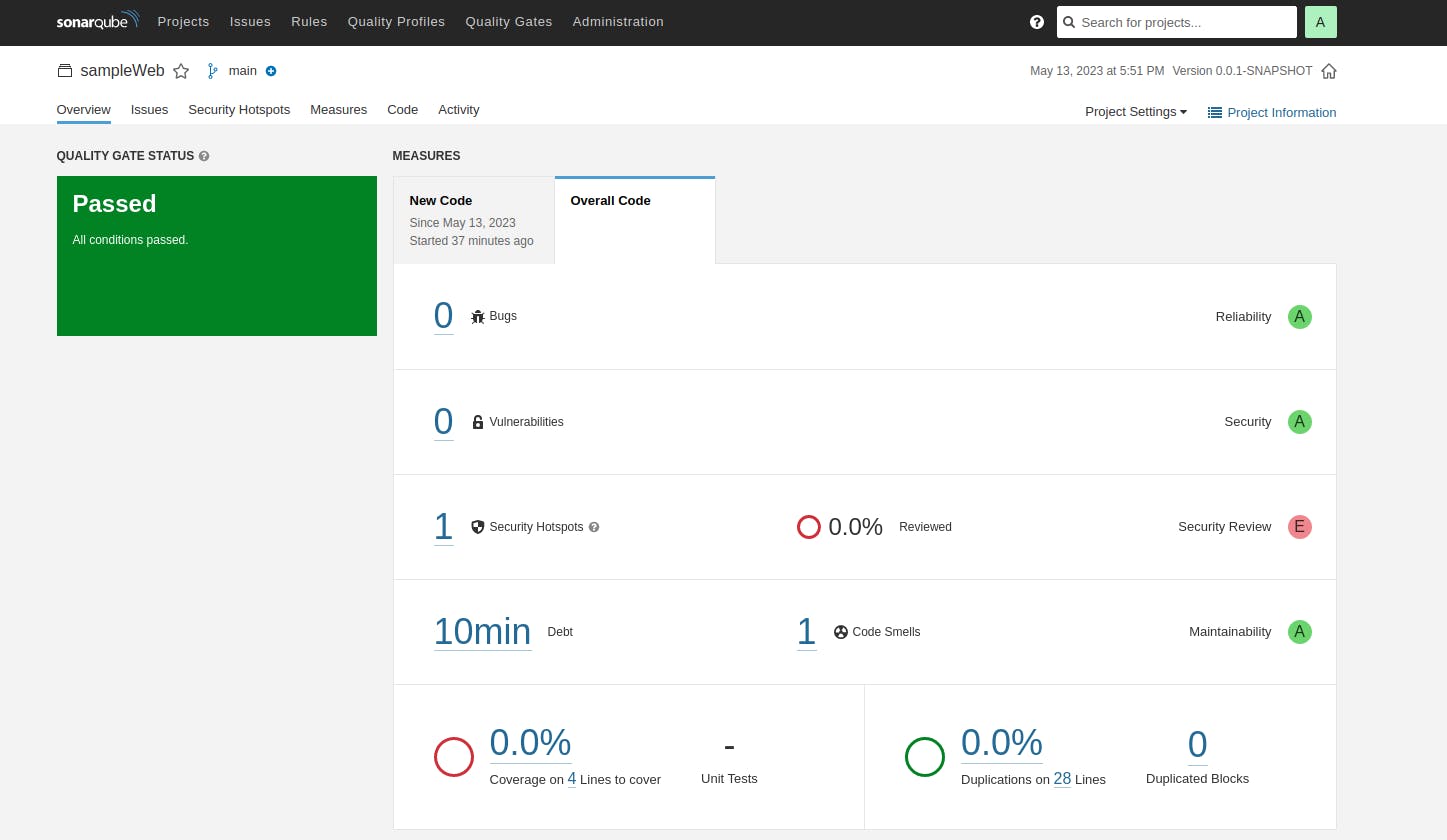
}

}

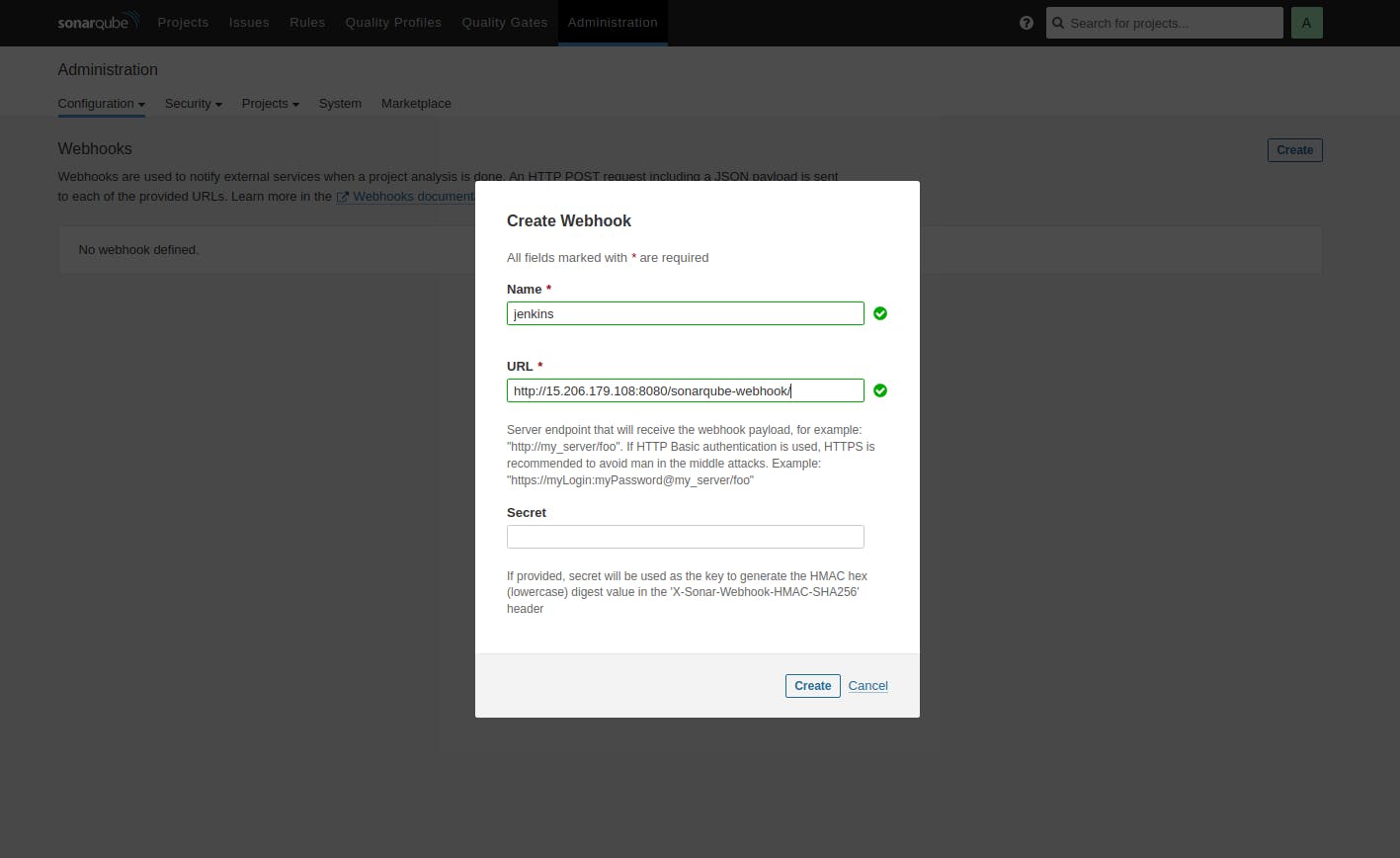
* We have give execute permission to gradlew and then we have executed ./gradlew sonarqube which helps us in pushing the code to sonarQube, where we will validate our checks against the sonar rules.
* Add the JenkinsFile to the git repo and push the changes.
* Go to the Jenkins dashboard and build the pipeline job and see if it builds successfully.



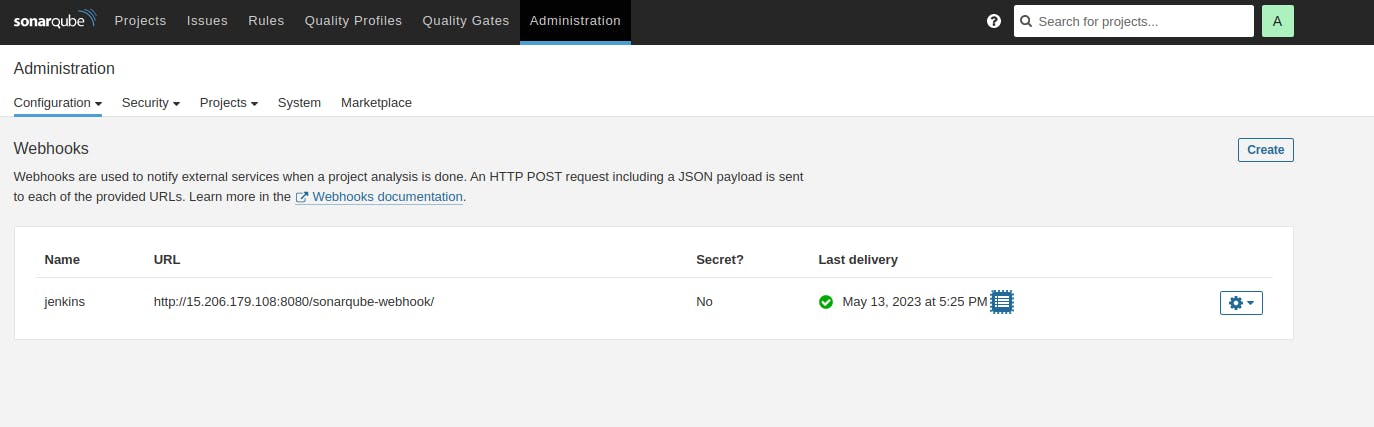
* We can see that the job was a success, we can also check the SonarQube Dashboard.



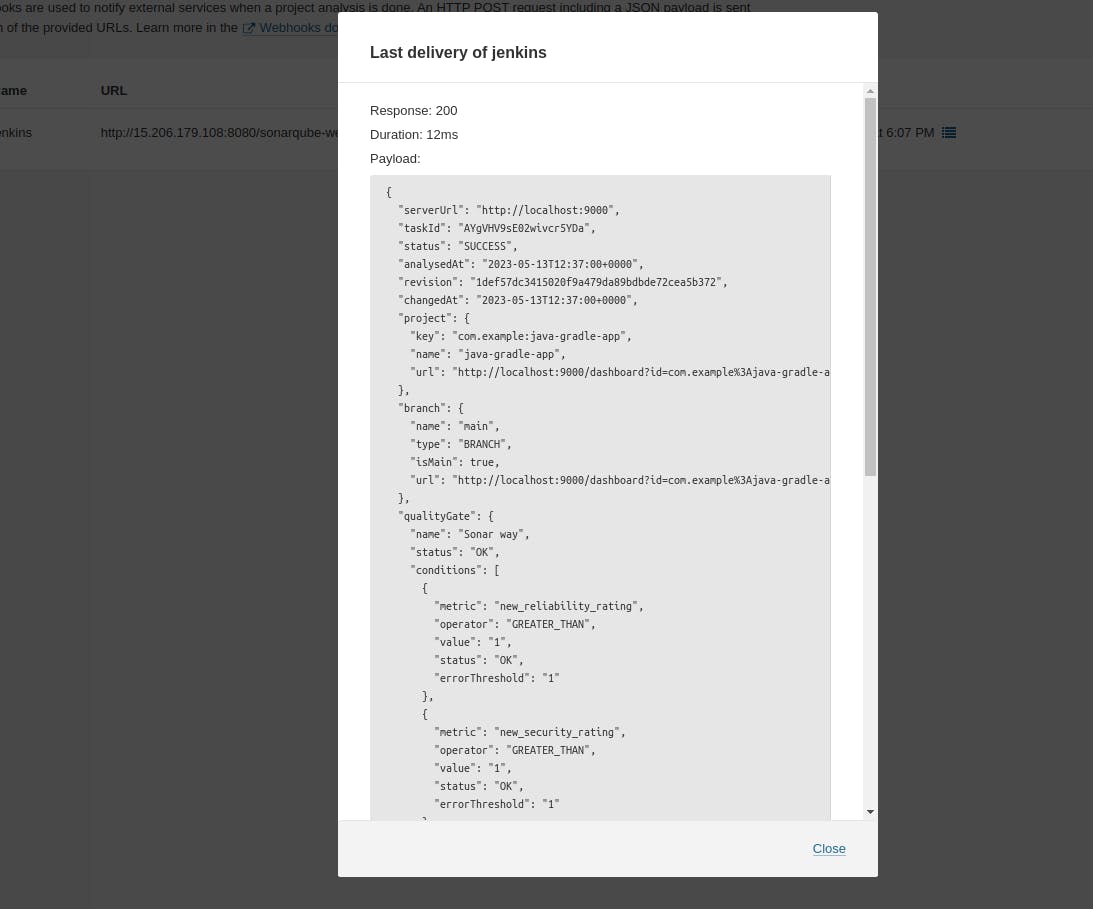
* Now, we will we also need to create a webhook to have connection between jenkins and sonarqube.
  1. To create a webhook navigate to sonarqube dashboard then administration > configuration > webhooks.
  2. Enter the URL of your jenkins host with port suffixed with  
     /sonarqube-webhook/ - The URL will be <http://jenkins_ip:8080/sonarqube-webhook/>



* The data that will be sent to jenkins via the sonarqube webhook can be seen by clicking the small icon beside the time in the Last delivery column (it will be available after we start a new build after creating the webhook).  
  Go back to jenkins dashboard and click on Build Now. Once the build is completed successfully, go back to SonarQube Dashboard and check the Last Delivery Data.



* We are interested in the qualityGate section of the Last Delivery data.



COPY

"qualityGate": {

"name": "Sonar way",

"status": "OK",

* If the status is ok then only our pipeline will move on to the next stage otherwise the build will fail.
* Now, we will add a block to our Sonar Quality Check stage which will wait for 15 minutes for the Quality Gate status to chaneg to ok state.

COPY

timeout(time: 15, unit: 'MINUTES') {

def qg = waitForQualityGate()

if (qg.status != 'OK') {

error "Pipeline aborted due to quality gate failure: ${qg.status}"

}

}

COPY

pipeline{

agent any

environment{

VERSION = "{env.BUILD\_ID}"

}

stages{

stage("Sonar Quality Check"){

steps{

script{

withSonarQubeEnv(credentialsId: 'sonar-token') {

sh 'chmod +x gradlew'

sh './gradlew sonarqube --info'

}

timeout(time: 15, unit: 'MINUTES') {

def qg = waitForQualityGate()

if (qg.status != 'OK') {

error "Pipeline aborted due to quality gate failure: ${qg.status}"

}

}

}

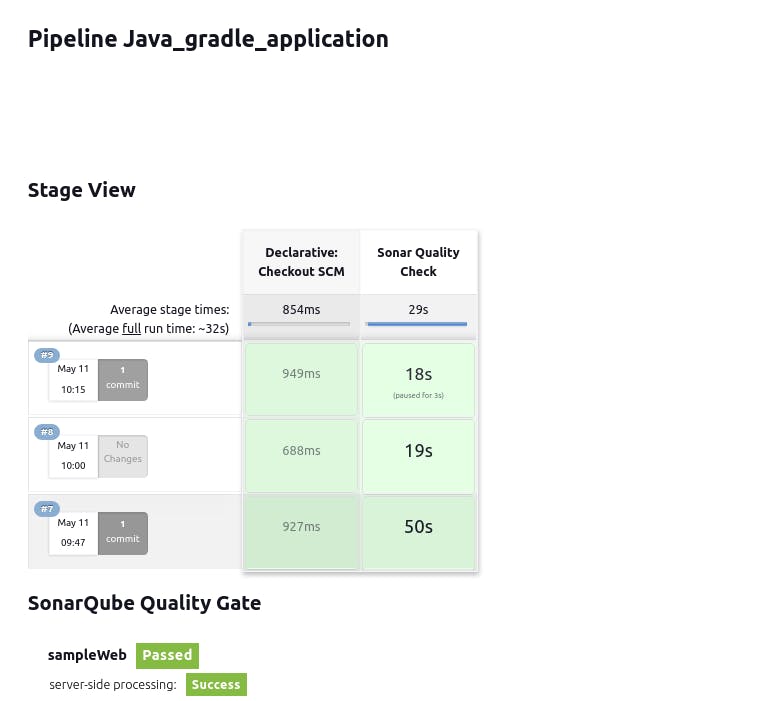
}

}

}

}

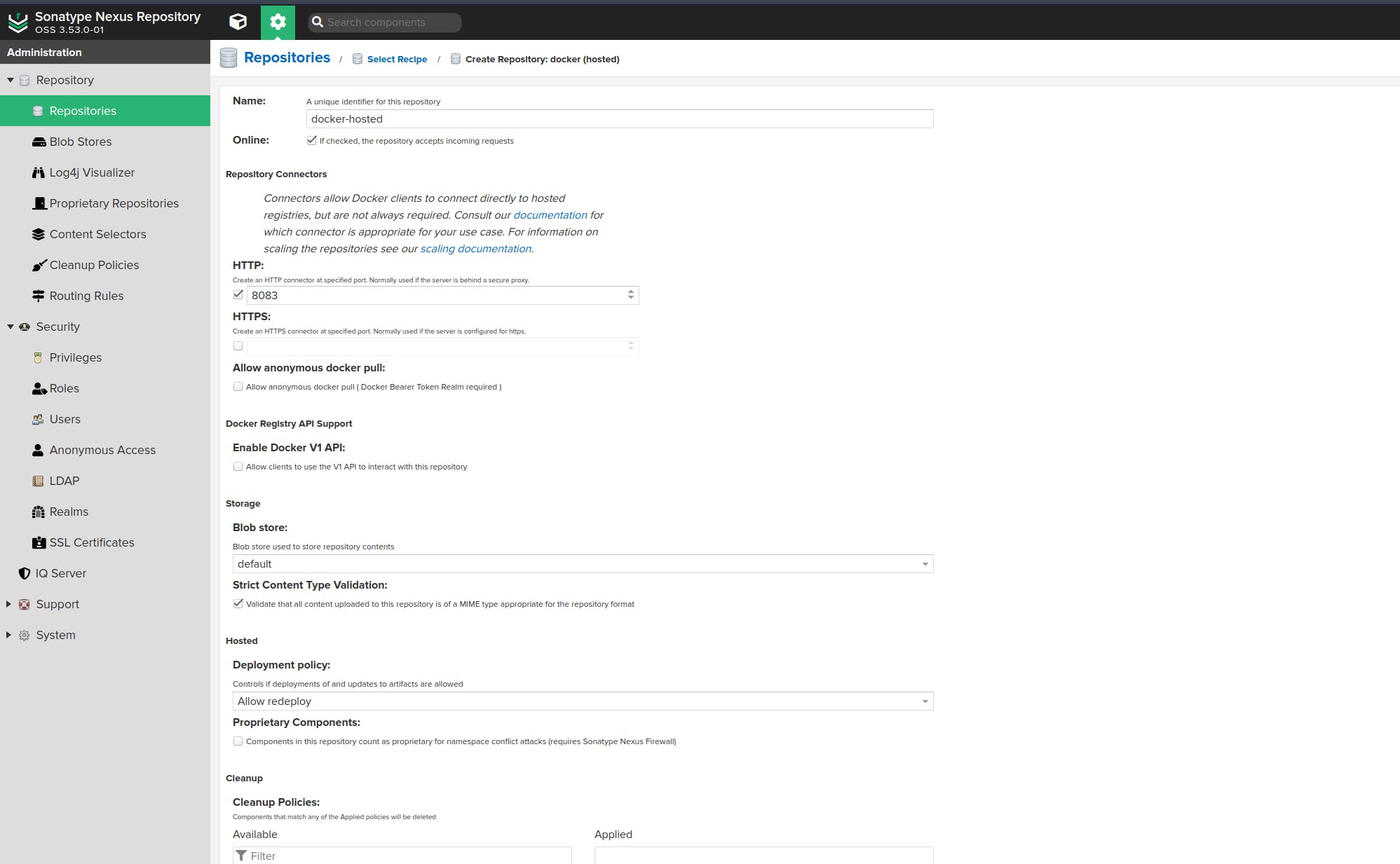
* Commit and push the changes to the dev branch and then run the build.



* This concludes the SonarQube Integration.

**STAGE II : Build docker images and push to Nexus**

1. **Create a private repository in Nexus Repository Manager**
   * We will create a private repository to store our docker images on the Nexus Repository Manager.
     + Go to Nexus dashboard > Repositories > Create Repository Select Recipe as docker-hosted Select HTTP port as 8083 Click on Create repository



* + - Next, we need to configure this repository on our jenkins server as an insecure registry to so that we can push our docker images to this repo.  
      To do so, go to the jenkins server and create or edit the file /etc/docker/daemon.json

COPY

vim /etc/docker/daemon.json

{ "insecure-registries":["nexus\_machine\_ip:8083"] }

It will look like this { "insecure-registries":["13.235.91.151:8083"] }

* + - Now restart the docker service using systemctl restart docker.service and check if the insecure registry is added properly.

COPY

root@jenkins:~# docker info | grep Insecure -A1

Insecure Registries:

13.235.91.151:8083

* + - We have successfully created and configured the nexus repository for storing our docker images, now let's try to login to the repository.

COPY

root@jenkins:~# docker login -u admin 13.235.91.151:8083

Password:

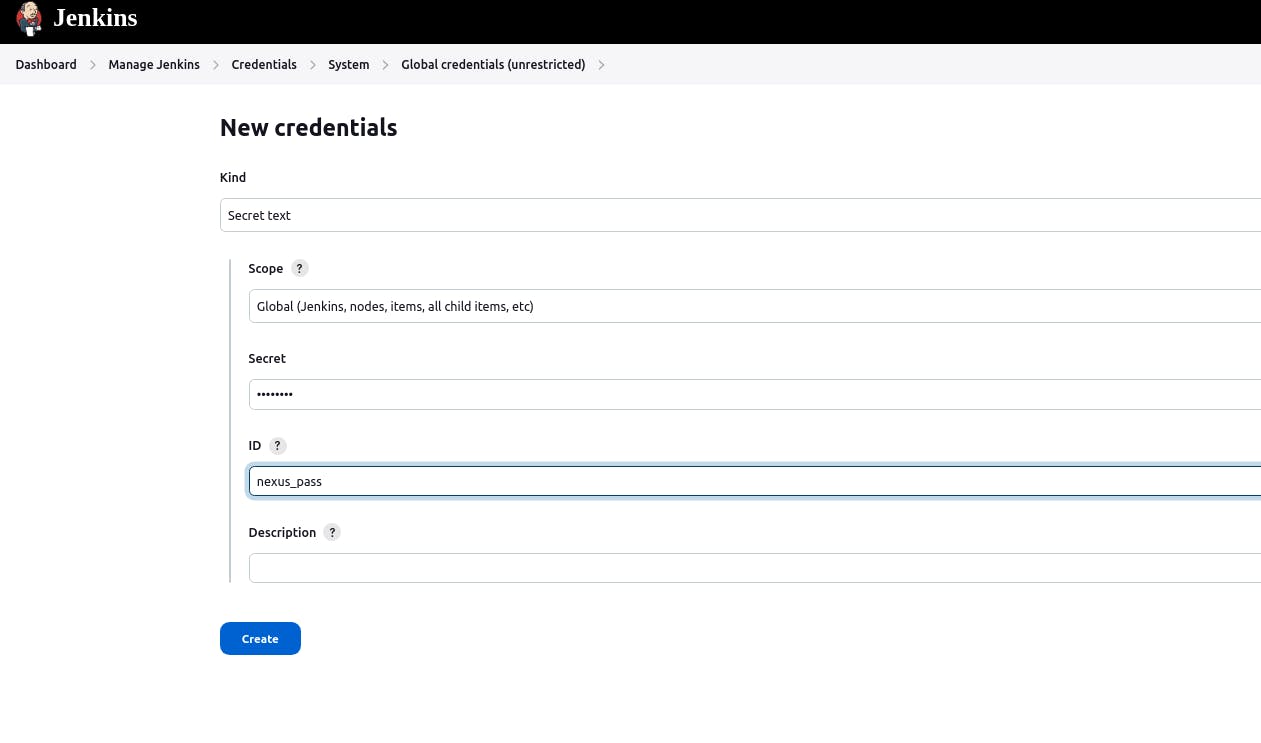
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.

Configure a credential helper to remove this warning. See

https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded

* + - Next, we need to add the password for our nexus repository as a secret in jenkins so that we can use it in our pipeline stages.  
      Goto Manage Jenkins > Security > Credentials. Click on global and then add credentials , select kind as secret text. Click on Create.



1. **Creating a multi-stage Dockerfile**
   * This Dockerfile is creating a Docker image that deploys a Java web application to Tomcat using Docker multi-stage build.

COPY

FROM openjdk:11 as base

WORKDIR /app

COPY . .

RUN chmod +x gradlew

RUN ./gradlew build

FROM tomcat:9

WORKDIR webapps/

COPY --from=base /app/build/libs/sampleWeb-0.0.1-SNAPSHOT.war .

RUN rm -rf ROOT && mv sampleWeb-0.0.1-SNAPSHOT.war ROOT.war

* + Here is a breakdown of what is happening in each step:
    - FROM openjdk:11 as base - This line sets the base image for the build process to the official OpenJDK 11 image.
    - WORKDIR /app - This line sets the working directory for the Docker container to /app.
    - COPY . . - This line copies the current directory (where the Dockerfile is located) into the /app directory in the Docker container.
    - RUN chmod +x gradlew - This line makes the gradlew script executable.
    - RUN ./gradlew build - This line runs the gradlew script to build the Java application.
    - FROM tomcat:9 - This line sets the base image to the official Tomcat 9 image for the second stage of our multi stage Dockerfile
    - WORKDIR webapps/ - This line sets the working directory for the Docker container to /usr/local/tomcat/webapps, where Tomcat looks for web applications.
    - COPY --from=base /app/build/libs/sampleWeb-0.0.1-SNAPSHOT.war . - This line copies the built Java web application from the "base" image to the current directory in the second image which is our final image.
    - RUN rm -rf ROOT && mv sampleWeb-0.0.1-SNAPSHOT.war ROOT.war - This line removes the existing ROOT web application and renames the copied application to ROOT.war so that it becomes the default application served by Tomcat.
  + We can check if our docker file is working as expected or not, go to the jenkins server and navigate to the path /var/lib/jenkins/workspace/sampleWeb\_app, here we will have all our project files from the git repo. We can create our Dockerfile here and try to build test images.

COPY

docker build -t test-tomcat .

jenkins@jenkins:~/workspace/java-gradle-app$ docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

test-tomcat latest 0589959663b8 25 minutes ago 514MB

* + Let's spin up a docker container from this image

COPY

docker run -itd -p 7777:8080 test-tomcat

jenkins@jenkins:~/workspace/java-gradle-app$ docker run -itd -p 7777:8080 test-tomcat

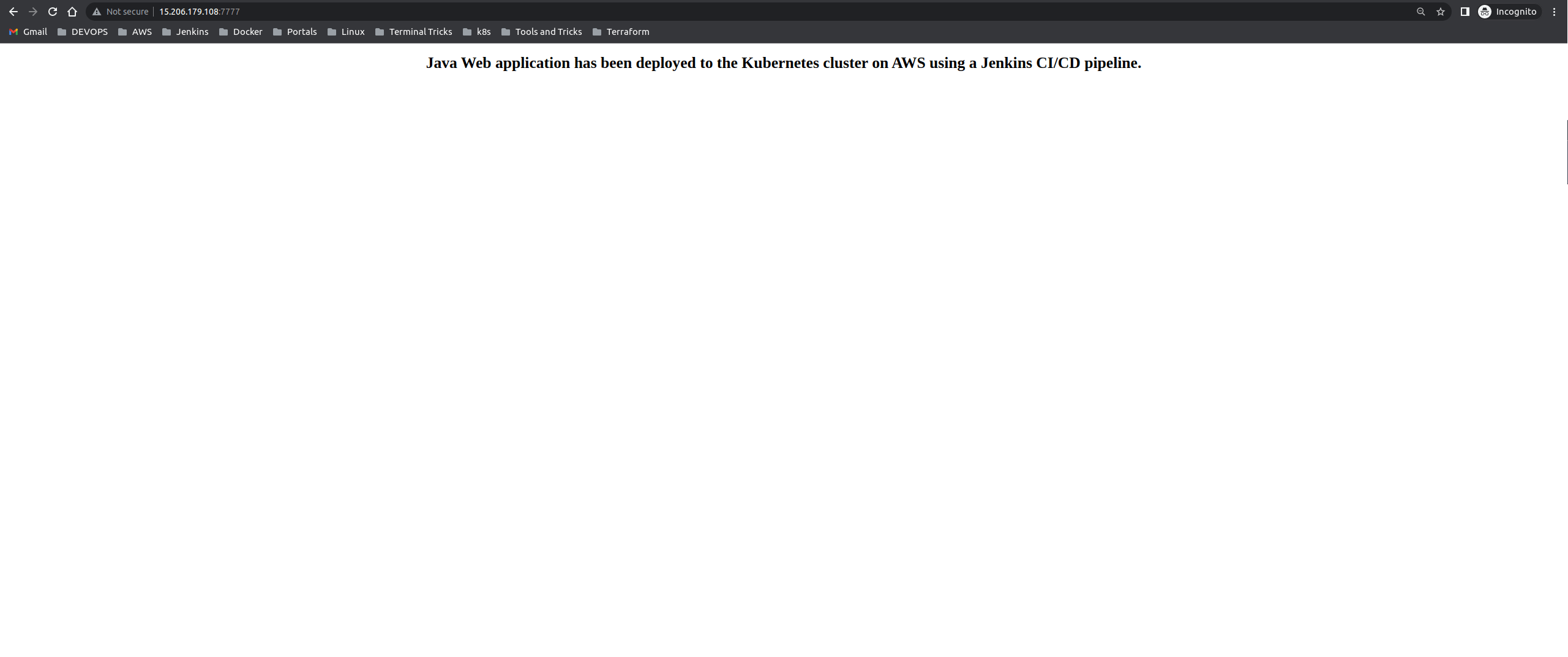
584f79d32cfa20f8b7428cdc9d2ca80928f717e14fb93ee59ee0bd7019ce2372

jenkins@jenkins:~/workspace/java-gradle-app$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

584f79d32cfa test-tomcat "catalina.sh run" 5 seconds ago Up 3 seconds 0.0.0.0:7777->8080/tcp, :::7777->8080/tcp nice\_mcclintock

* + We can access the web application by going to jenkins\_ip:7777 in our web browser.



* + Make sure to delete the containers and images we created for our testing purpose in the previous test after confirming that our container is running as expected

1. **Adding Stage II : Build docker images and push to Nexus to Jenkinsfile**
   * We need to do four steps to build our image and push the image to the nexus repository.
     + **Tag and build docker image**
       - We can tag our image using the command  
         docker build -t nexus\_server\_ip:8083/myapp:$VERSION
       - VERSION needs to be unique as every change to the application will trigger a new job and will create a new image so we need a variable which we can use as tag, we can use the build number of our jenkins job as a tag. The image created by the job will have the build number as tag.
       - For this to work, first we need to define $VERSION in the pipeline, it's value will be build number of the jenkins job.
       - Define the VERSION in the pipeline using environment variable which will be availabe for use during the execution of the pipeline.

COPY

environment{

VERSION = "${env.BUILD\_ID}"

}

* + - **Login to the Nexus repo**
      * This can be done using the command  
        docker login -u admin -p $nexus\_pass\_var nexus\_server\_ip:8083
      * $nexus\_pass\_var is variable through which we access the jenkins credential that we created for the storing the admin password of the nexus repository.
      * We will use a withCredentials block to access the credential nexus\_pass.

COPY

withCredentials([string(credentialsId: 'nexus\_pass', variable: 'nexus\_pass\_var')]) {

sh '''

docker build -t nexus\_server\_ip:8083/myappapp:${VERSION} .

docker login -u admin -p $nexus\_docker\_repo\_pass\_var nexus\_server\_ip:8083

* + - **Push the docker image to the nexus repo**

COPY

#Use this command to push the image to nexus repo

docker push nexus\_server\_ip:8083/myappapp:${VERSION}

* + - Remove the image from the server after the image is pushed to nexus

COPY

docker rmi nexus\_server\_ip:8083/myapp:${VERSION}

* + The final code of this stage of pipeline will look like this:  
    We've also created a new environment variable in the pipeline called DOCKER\_HOSTED\_EP which will declare the value of nexus\_machine\_ip:8083 as an variable which will be available to the pipeline through all the stages.

COPY

pipeline{

agent any

environment{

VERSION = "${env.BUILD\_ID}"

DOCKER\_HOSTED\_EP = "13.235.91.151:8083"

}

stages{

stage("Sonar Quality Check"){

steps{

script{

withSonarQubeEnv(credentialsId: 'sonar-token') {

sh 'chmod +x gradlew'

sh './gradlew sonarqube --info'

}

timeout(time: 1, unit: 'MINUTES') {

def qg = waitForQualityGate()

if (qg.status != 'OK') {

error "Pipeline aborted due to quality gate failure: ${qg.status}"

}

}

}

}

}

stage("Build docker images and push to Nexus"){

steps{

script{

withCredentials([string(credentialsId: 'nexus\_pass', variable: 'nexus\_pass\_var')]) {

sh '''

docker build -t $DOCKER\_HOSTED\_EP/javawebapp:${VERSION} .

docker login -u admin -p $nexus\_pass\_var $DOCKER\_HOSTED\_EP

docker push $DOCKER\_HOSTED\_EP/javawebapp:${VERSION}

docker rmi $DOCKER\_HOSTED\_EP/javawebapp:${VERSION}

'''

}

}

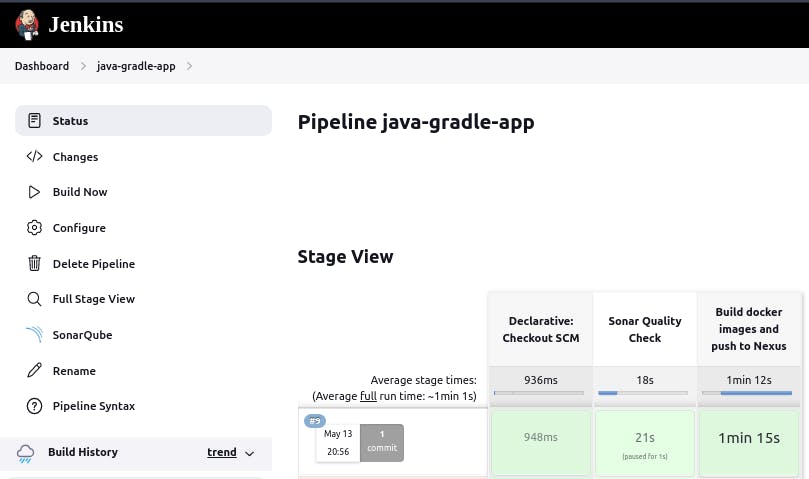
}

}

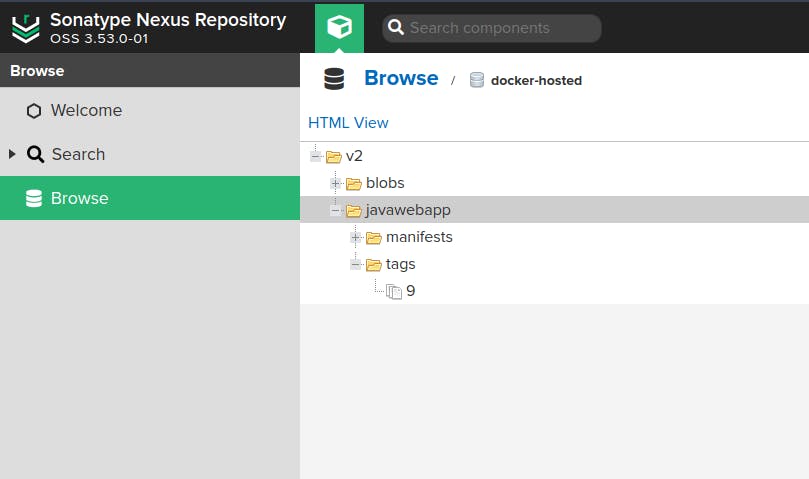
}

}

* + Once the build is successfully completed, check the nexus repository  
    docker-hosted to see if the docker images were pushed to it.



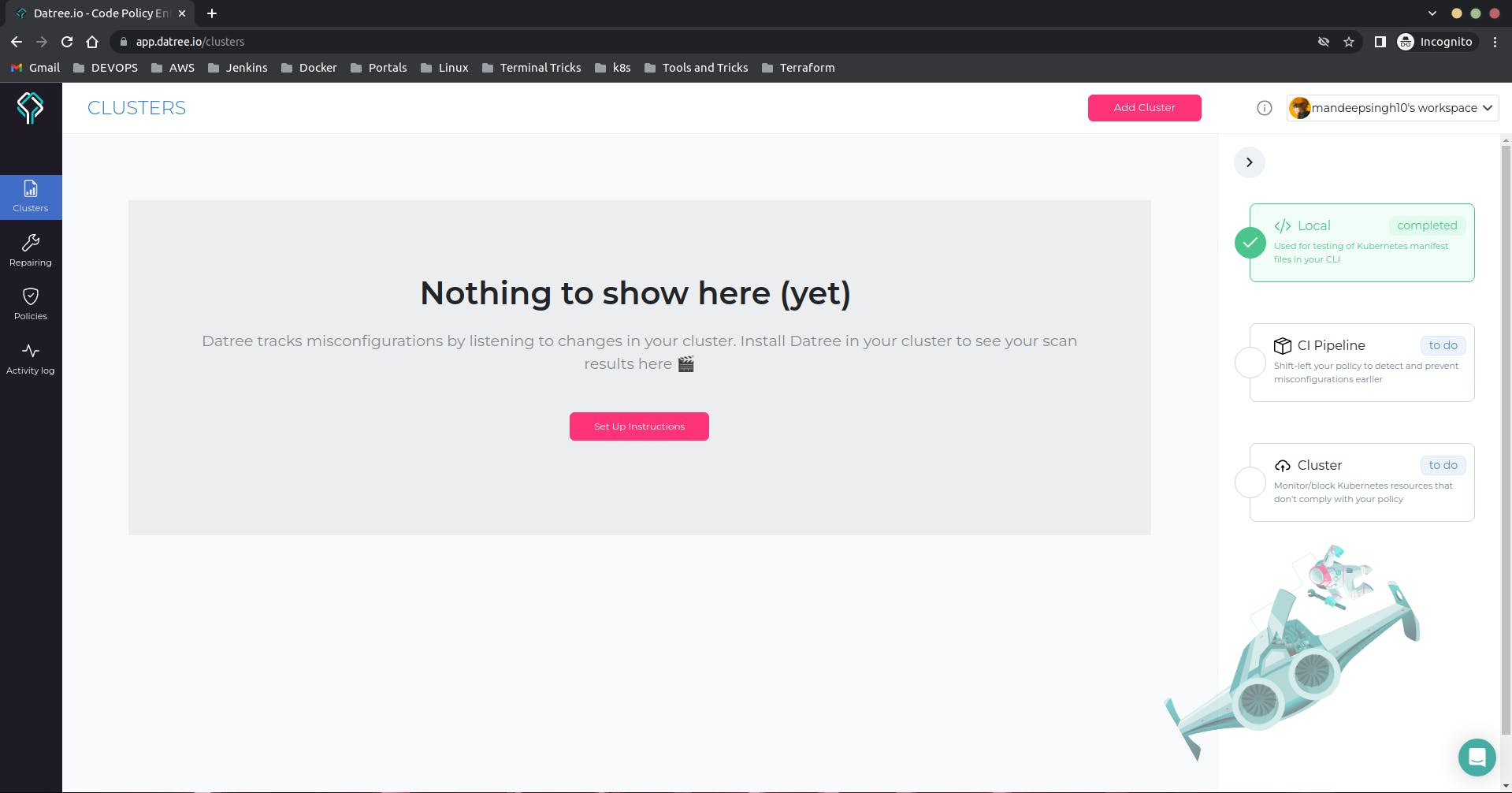
* + Build was successfull, build number is **9**, let's see if we have docker images in the nexus repository with tag as **9**.



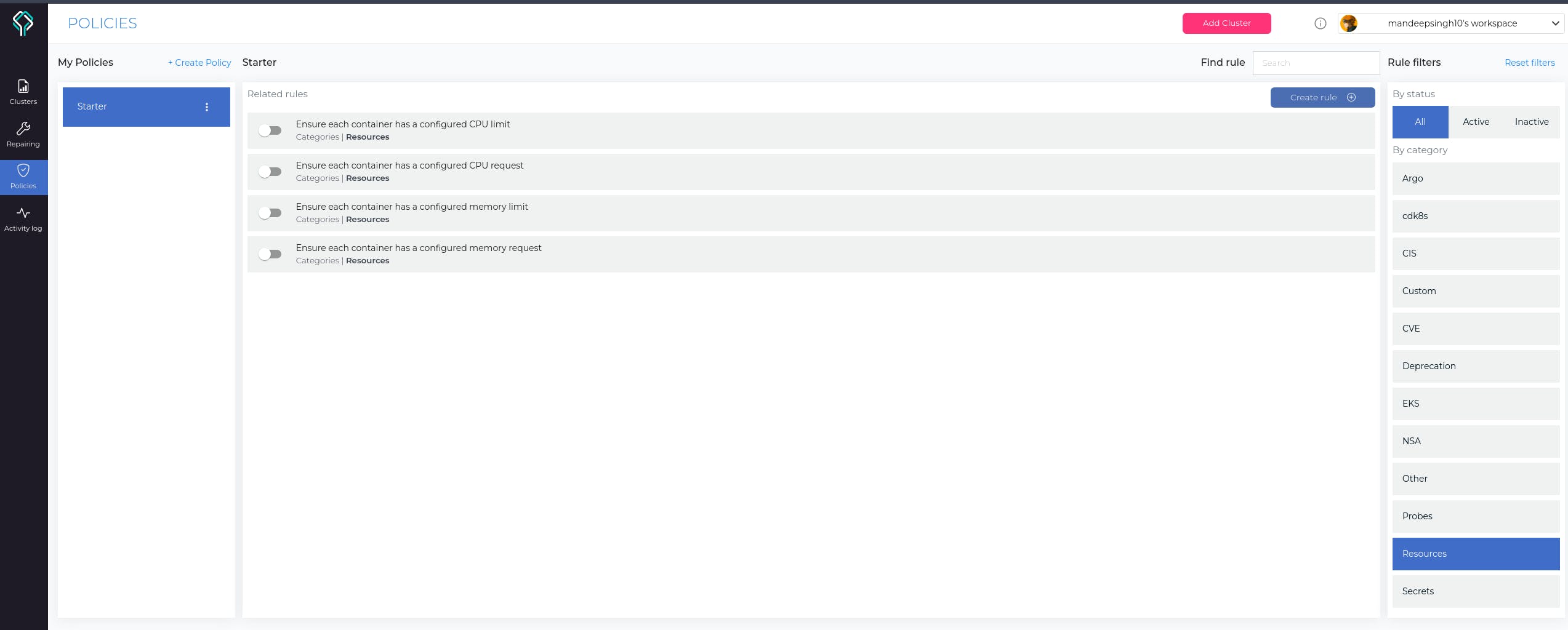
* + **Docker image successfully pushed to the nexus repository!!!!!**
  + This concludes Stage II of our Pipeline, let's move on to Stage III.

**STAGE III : Identify misconfigurations in Helm charts using**[datree.io](http://datree.io/)

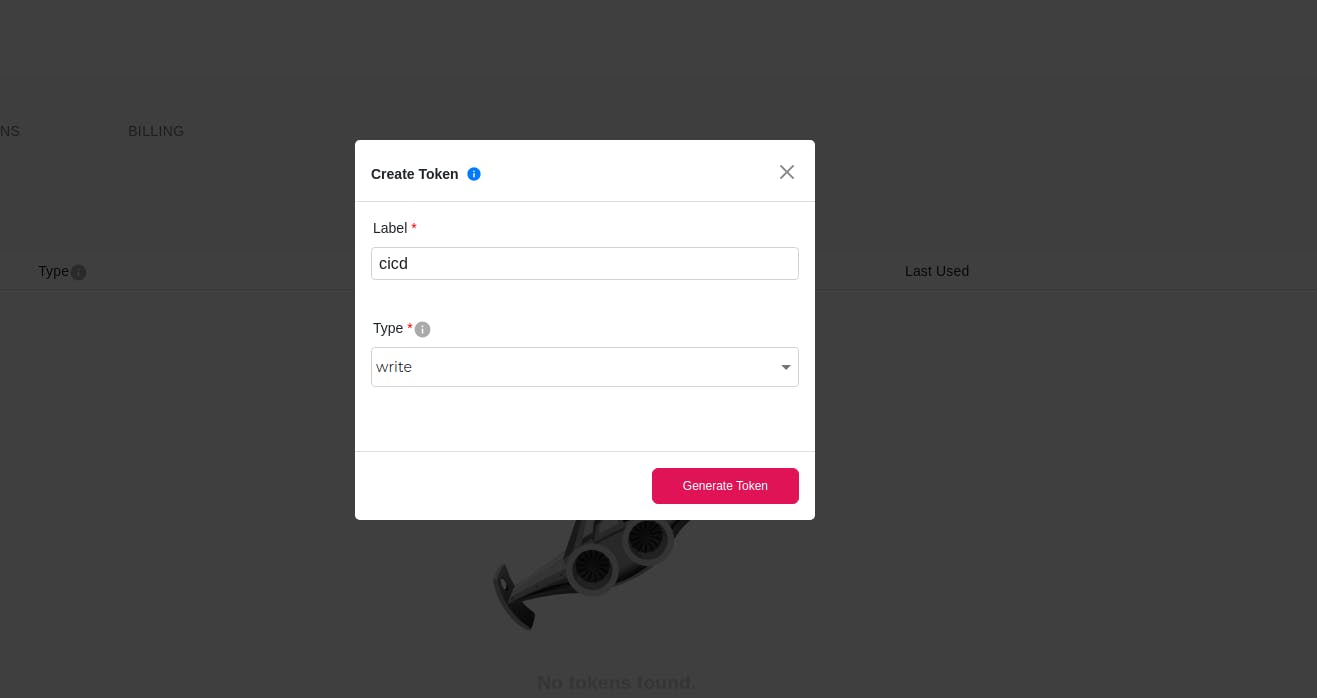
* In this stage we will identify the misconfigurations in our HELM charts.
* We have already created the required YAML files for deployments, services and helm charts. These are present in the kubernetes/myapp folder in the root directory of the repo.
* First we will need a token from our [datree.io](http://datree.io/) account in start analzing helm charts based on the policies/rules set in our [datree.io](http://datree.io/) account.
  1. Go to [datree.io](http://datree.io/) and login using your github or gmail account, after successful login, we will see the [datree.io](http://datree.io/) dashboard



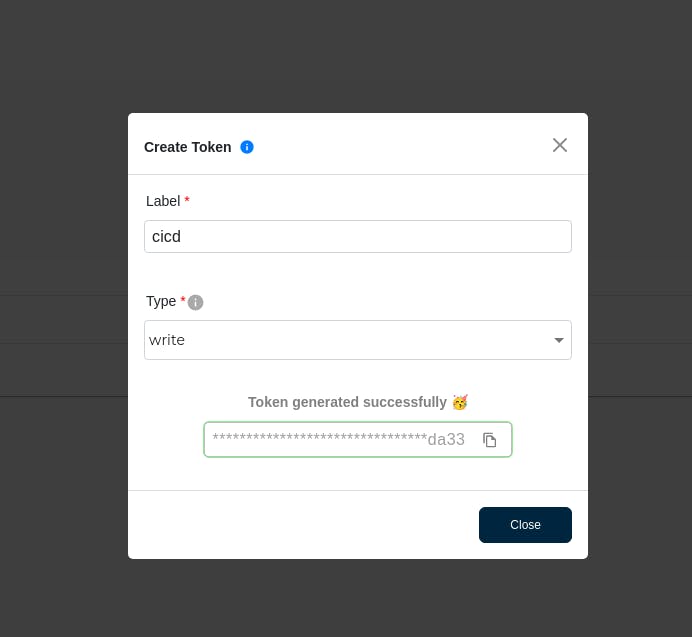
* 1. The policies/rules against which our helm chart will be analzed can se accesed by navigating to **Policies > Active Rules.**



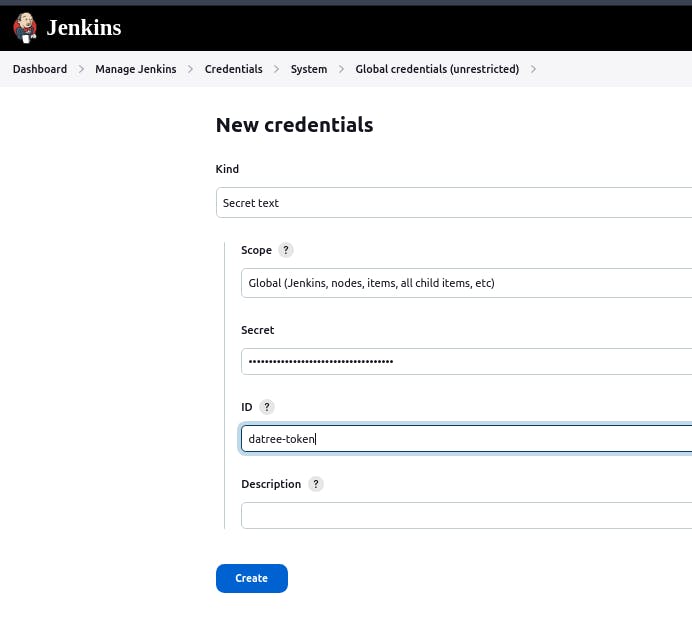
* 1. Next, go to **SETTINGS > TOKEN MANAGEMENT,** click on Create Token



Enter a name in the Label section and select write in the Type  
section, click on Generate Token.



* 1. Copy the token and create a Jenkins credential datree-token of Kind Secret text, enter the token as secret, click on Create.



* 1. Our helm charts are in the kubernetes/myapp directory so we have to perform three steps sop that we can do static code analysis of our helm charts using datree.
     1. Change the working directory to kubernetes/

COPY

dir('kubernetes/') {

}

* + 1. Set datree token from the value of jenkins secret credential datree-token.

COPY

dir('kubernetes/') {

withCredentials([string(credentialsId: 'datree-token', variable: 'datree\_token\_var')]) {

sudo helm datree config set token $datree\_token\_var

}

}

* + 1. Run the command helm datree test myapp/

COPY

dir('kubernetes/') {

withCredentials([string(credentialsId: 'datree-token', variable: 'datree\_token\_var')]) {

sh '''

sudo helm datree config set token $datree\_token\_var

sudo helm datree test myapp/

'''

}

}

* + 1. Now, the complete pipeline for this stage will look like this:

COPY

stage('Identifying the misconfiguration in HELM charts using datree'){

steps{

script{

dir('kubernetes/') {

withCredentials([string(credentialsId: 'datree-token', variable: 'datree\_token\_var')]) {

sh '''

sudo helm datree config set token $datree\_token\_var

sudo helm datree test myapp/

'''

}

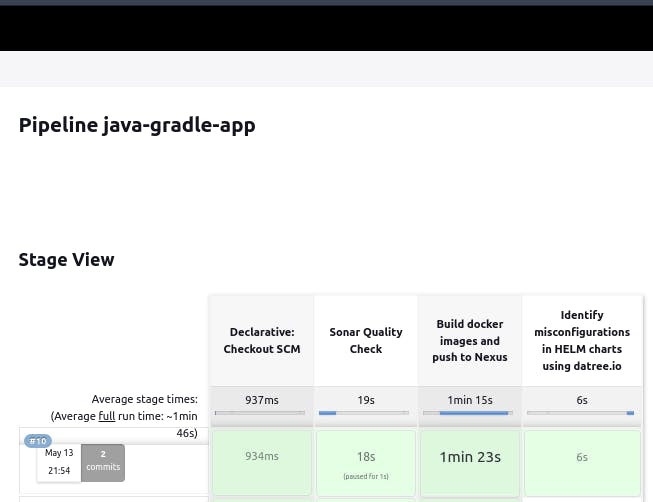
}

}

}

}

* + 1. Next, we wil edit the jenkins file and commit the changes to the dev branch and then start a new build.



* + 1. Build was successful, let's check the console output for more details.

COPY

Pipeline] { (Identify misconfigurations in HELM charts using datree.io)

[Pipeline] script

[Pipeline] {

[Pipeline] dir

Running in /var/lib/jenkins/workspace/java-gradle-app/kubernetes

[Pipeline] {

[Pipeline] withCredentials

Masking supported pattern matches of $datree\_token\_var

[Pipeline] {

[Pipeline] sh

+ sudo helm datree config set token \*\*\*\*

+ sudo helm datree test myapp/

[K

| Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

/ Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

- Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

\ Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

| Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

/ Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

- Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

\ Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

| Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

/ Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

- Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

\ Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

| Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

/ Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

- Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

\ Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

| Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

/ Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

- Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

\ Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

| Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

/ Loading... WWWWWWWWWWWWWW[K[K[K[K[K[K[K[K[K[K[K[K[K[K

[K

(Summary)

- Passing YAML validation: 1/1

- Passing Kubernetes (1.24.0) schema validation: 1/1

- Passing policy check: 1/1

+-----------------------------------+-----------------------+

| Enabled rules in policy "Starter" | 0 |

| Configs tested against policy | 2 |

| Total rules evaluated | 0 |

| [36mTotal rules skipped[0m | [36m0[0m |

| [91mTotal rules failed[0m | [91m0[0m |

| [32mTotal rules passed[0m | [32m0[0m |

| See all rules in policy | https://app.datree.io |

+-----------------------------------+-----------------------+

[Pipeline] }

[Pipeline] // withCredentials

[Pipeline] }

[Pipeline] // dir

[Pipeline] }

[Pipeline] // script

[Pipeline] }

[Pipeline] // stage

[Pipeline] }

[Pipeline] // withEnv

[Pipeline] }

[Pipeline] // withEnv

[Pipeline] }

[Pipeline] // node

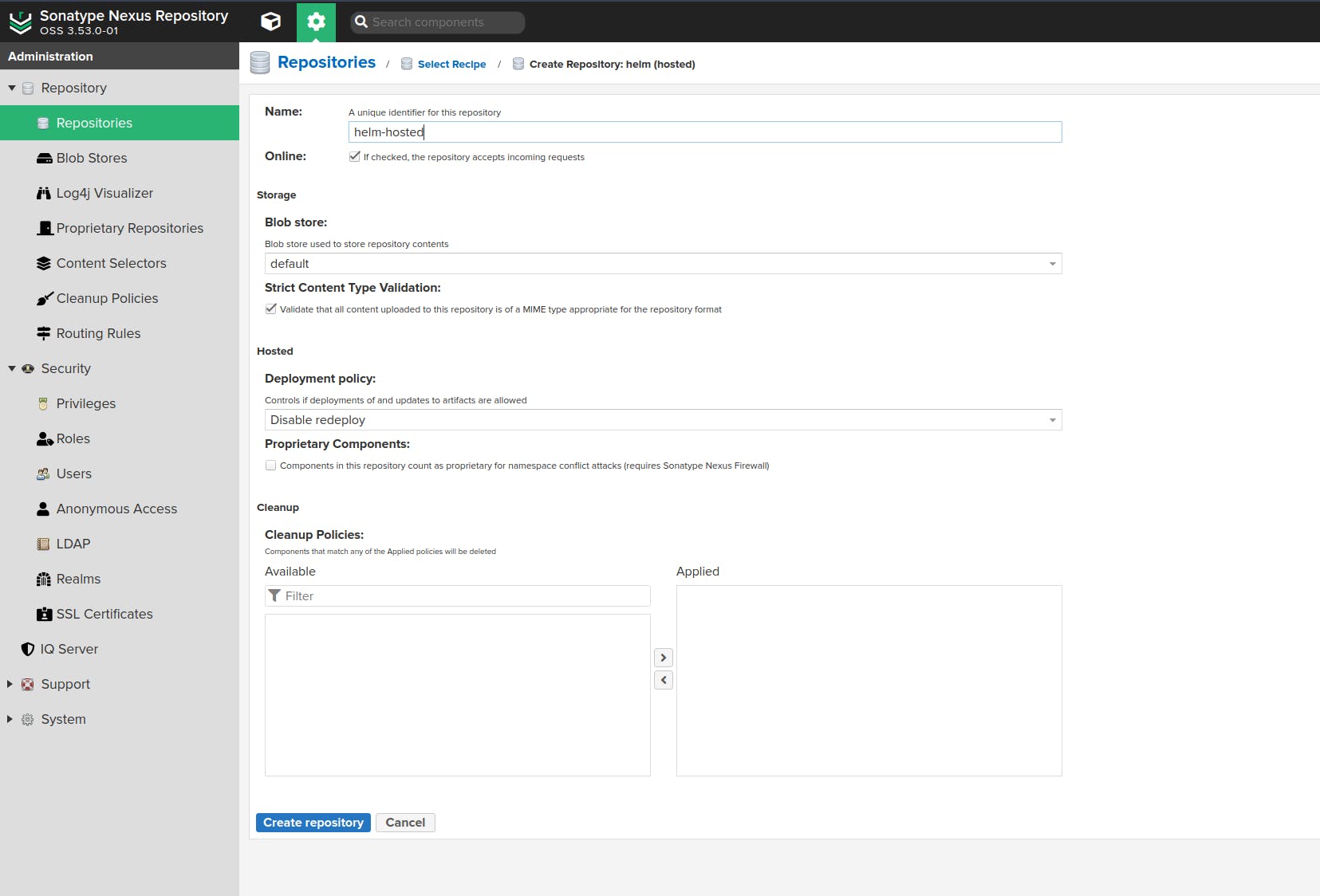
[Pipeline] End of Pipeline

Finished: SUCCESS

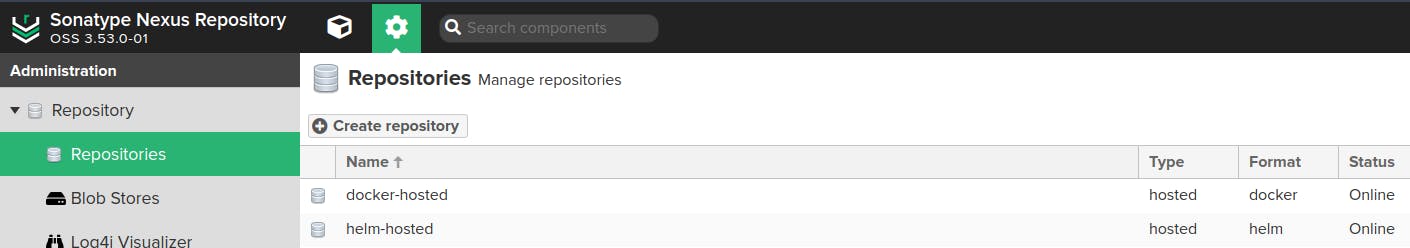
* + 1. We can see that datree plugin worked as expected. This concludes Stage III.

**STAGE IV : Push Helm Charts to Nexus Repo**

1. **Create a private repository in Nexus Repository Manager**
   * We will create a private repository to store our Helm charts on the Nexus Repository Manager and
     + Go to Nexus dashboard > Repositories > Create Repository Select Recipe as helm-hosted. Click on Create repository.



* + - We can see our helm-hosted repository listed under repositories.



* + - Next step is to create the Stage in Jenkins Pipeline for pushing the helm charts to Nexus repo, for this no other configuration is need in Jenkins host because we will use Nexus repo api to push the helm charts.

COPY

curl -u admin:$nexus\_pass\_var http://nexus\_machine\_ip:8081/repository/helm-hosted/ --upload-file myapp-${helmchartversion}.tgz -v

* + - We will create a new environment variable in the pipeline called HELM\_HOSTED\_EP which will replace hard coded value for nexus\_machine\_ip:8081 same as when we pushed the docker images and used the environment variable DOCKER\_HOSTED\_EP.
    - Next, we need to package our helm according to the helm chart version, this can be done by simply running the command helm package myapp.

COPY

jenkins@jenkins:~/workspace/java-gradle-app/kubernetes$ cat myapp/Chart.yaml | grep 'version:'

version: 0.1.0

jenkins@jenkins:~/workspace/java-gradle-app/kubernetes$ helm package myapp/

Successfully packaged chart and saved it to: /var/lib/jenkins/workspace/java-gradle-app/kubernetes/myapp-0.1.0.tgz

* + - Helm package command will package the helm charts accoding to the chart version mentioned in myapp/Chart.yaml.
    - Our final code for this stage of the pipeline will look like this

COPY

environment{

VERSION = "${env.BUILD\_ID}"

DOCKER\_HOSTED\_EP = "13.235.91.151:8083"

HELM\_HOSTED\_EP = "13.235.91.151:8081"

}

...

...

stage("Push Helm Charts to Nexus Repo"){

steps{

script{

dir('kubernetes/'){

withCredentials([string(credentialsId: 'nexus\_pass', variable: 'nexus\_pass\_var')]) {

sh '''

helmchartversion=$(helm show chart myapp/ | grep version | awk '{print $2}')

helm package myapp/

curl -u admin:$nexus\_pass\_var http://$HELM\_HOSTED\_EP/repository/helm-hosted/ --upload-file myapp-${helmchartversion}.tgz -v

'''

}

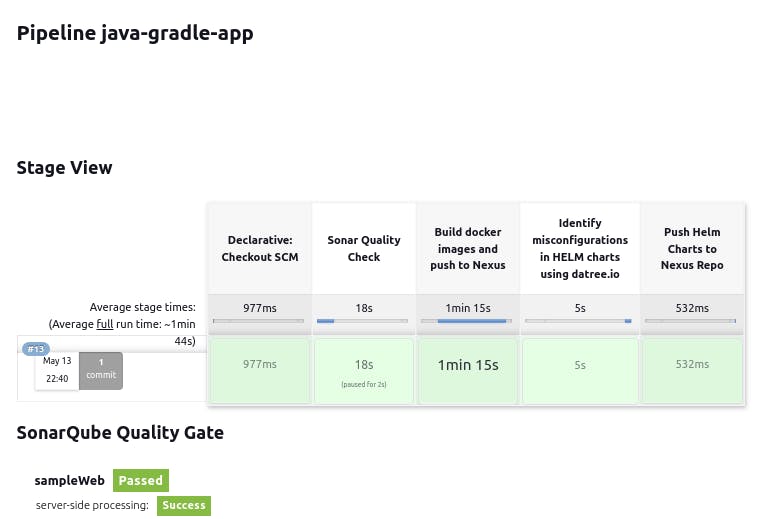
}

}

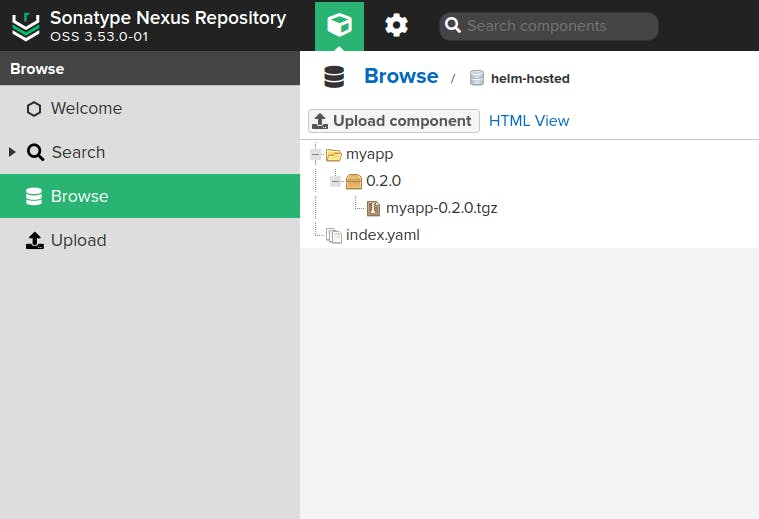
}

}

* + - Let's run the build now.



* + - The build was successful, now let's check if there were any images pushed to the helm repository.



* + - Helm Charts pushed to the helm-hosted Nexus repo with the same tag as the helm chart version.
    - This concludes the Stage IV of our pipeline.

**Stage V: Deploy application on k8s cluster**

* To deploy our applicatiojn on k8s-cluster we need to perform some additional steps.
  1. **Configuring the Jenkins servers to access the kubernetes cluster and run administrative commands.**
     + This was already done by the magic of our ansible playbooks, the requirement was to install kubectl utility on the Jenkins node (implemented in jenkins.yaml) and copy the the kubeconfig file i.e /etc/kubernetes/admin.conf or /root/.kube/config, both of them are the same.
     + We've copied the /root/.kube/config file from the k8s-master node to the $HOME directory of the jenkins user /var/lib/jenkins/.kube/config on the Jenkins node.
     + These steps were performed by these plays in the k8s\_cluster\_setup.yaml.

COPY

#### Plays to copy kubeconfig to jenkins server for jenkins user #### Required for Stage V & VI : Deploying application on k8s cluster #####

##### PLAY 1 : COPY kubeconfig from k8s-master node to localhost #####

- name: Copy the kubeconfig from k8s-master

hosts: k8s-master

become: true

tasks:

- name: Fetch the file from the k8s-master to localhost

run\_once: yes

fetch:

src: /root/.kube/config

dest: buffer/

flat: yes

##### PLAY 2 : INSTALL KUBECTL, CREATE .kube direcoty in $HOME of jenkins user, copy kubeconfig from localhost buffer to Jenkins server and sets the permission to 0600 and ownership to jenkins:jenkins#####

- name: Jenkins User kubectl Setup

hosts: jenkins

become: true

tasks:

- name: Install kubectl

snap:

name: kubectl

classic: true

state: present

- name: Create directory and set ownership of .kube directory for jenkins user

file:

path: /var/lib/jenkins/.kube

state: directory

owner: jenkins

group: jenkins

- name: Copy the file from localhost to jenkins

copy:

src: buffer/config

dest: /var/lib/jenkins/.kube/config

mode: "0600"

owner: jenkins

group: jenkins

* + - We decided to add these plays to the k8s\_cluster\_setup.yaml instead of a standalone Ansible playbook because they have a dependency on the K8s cluster being set up.
    - If we created a standalone playbook and someone tried to run it before the cluster was provisioned, they would encounter errors. By including the plays in the k8s\_cluster\_setup.yaml, we ensure that all necessary configurations for the Jenkins node to communicate with the K8s cluster are completed after the cluster is provisioned.
    - We can now sit back and relax knowing that once the cluster is provisioned in the earlier stages, everything is set up correctly.
  1. **Adding the docker-hosted Nexus repository on both the k8s cluster nodes**
     + We are using containerd (CRI) so we cannot install docker runtime and configure insecure repo for docker as it will cause many issues in our cluster.
     + We will have to configure our nexus repo as an insecure repository for containerd. The below mentioned steps need to be performed on both the k8s-nodes (we are only scheduling our deployments on k8s-node1 but it's a good practice to keep the configuration consisitent so we will do it for both the nodes).
       1. Set the default config file for **container.d** as /etc/containerd/config.toml

COPY

mkdir -p /etc/containerd

containerd config default>/etc/containerd/config.toml

systemctl restart containerd

systemctl status containerd.service

* + - 1. Edit the /etc/containerd/config.toml
         * Find the line [plugins."io.containerd.grpc.v1.cri".registry.configs]
         * Add these six lines below it

COPY

[plugins."io.containerd.grpc.v1.cri".registry.configs."13.235.91.151:8083"]

[plugins."io.containerd.grpc.v1.cri".registry.configs."13.235.91.151:8083".tls]

ca\_file = ""

cert\_file = ""

insecure\_skip\_verify = true

key\_file = ""

Our docker-hosted repository is at 13.235.91.151:8083 so we are using this IP:PORT combination. Replace the IP address with the public ip of the Nexus Repository Manager and if you specified a different port for docker-hosted while confguring then use that.

* + - * + Find the line [plugins."io.containerd.grpc.v1.cri".registry.mirrors]
        + Add these two lines below it

COPY

[plugins."io.containerd.grpc.v1.cri".registry.mirrors."13.235.91.151:8083"]

endpoint = ["http://13.235.91.151:8083"]

* + - * + It will look like this after all the changes.

COPY

[plugins."io.containerd.grpc.v1.cri".registry.configs] [plugins."io.containerd.grpc.v1.cri".registry.configs."13.235.91.151:8081"] [plugins."io.containerd.grpc.v1.cri".registry.configs."13.235.91.151:8081".tls]

ca\_file = ""

cert\_file = ""

insecure\_skip\_verify = true

key\_file = "" [plugins."io.containerd.grpc.v1.cri".registry.headers] [plugins."io.containerd.grpc.v1.cri".registry.mirrors]

[plugins."io.containerd.grpc.v1.cri".registry.mirrors."13.235.91.151:8083"]

endpoint = ["http://13.235.91.151:8083"]

* + - * + Restart containerd service and try pulling an image from the docker-hosted Nexus repo.

COPY

root@k8s-master:~# systemctl restart containerd.service

root@k8s-master:~# crictl pull 13.235.91.151:8083/javawebapp:11

Image is up to date for sha256:603cc7300cb07bf4ec156ddfcdaa72698fbeb441bda60bbc84704505e3c1428e

root@k8s-master:~# crictl images | grep javawebapp

13.235.91.151:8083/javawebapp 11 603cc7300cb07 287MB

* + - * + We are able to pull the container images from our private nexus repository docker-hosted.
        + Repeat the steps on the k8s-node1, our application will be deployed on the worker node as by default k8s control plane node has a taint on it which forbids us to deploy any pods on it except the kube-system ones.  
          Taints: node-role.kubernetes.io/control-plane:NoSchedule
        + Once these steps are completed on the k8s worker node as well then we are good to go.
    - **Configure Mail Server and add post block in Jenkins Pipeline**
      1. Goto Manage Jankins > Manage Plugins > Installed and make sure that the Email Extension Plugin is installed.
      2. Goto Manage Jankins > Configure Systems > Email Notification Let's configure smtp config for our gmail account and try sending a test email.



* + - 1. To configure these setting we'll need to go to our gmail account settings > Manage Your Google Account. Now we need to create an app password so that our jenkins app can send emails to our gmail account.
      2. **Important:** To create an app password, you need 2-Step Verification on your Google Account.

If you use 2-Step-Verification and get a "password incorrect" error when you sign in, you can try to use an app password.

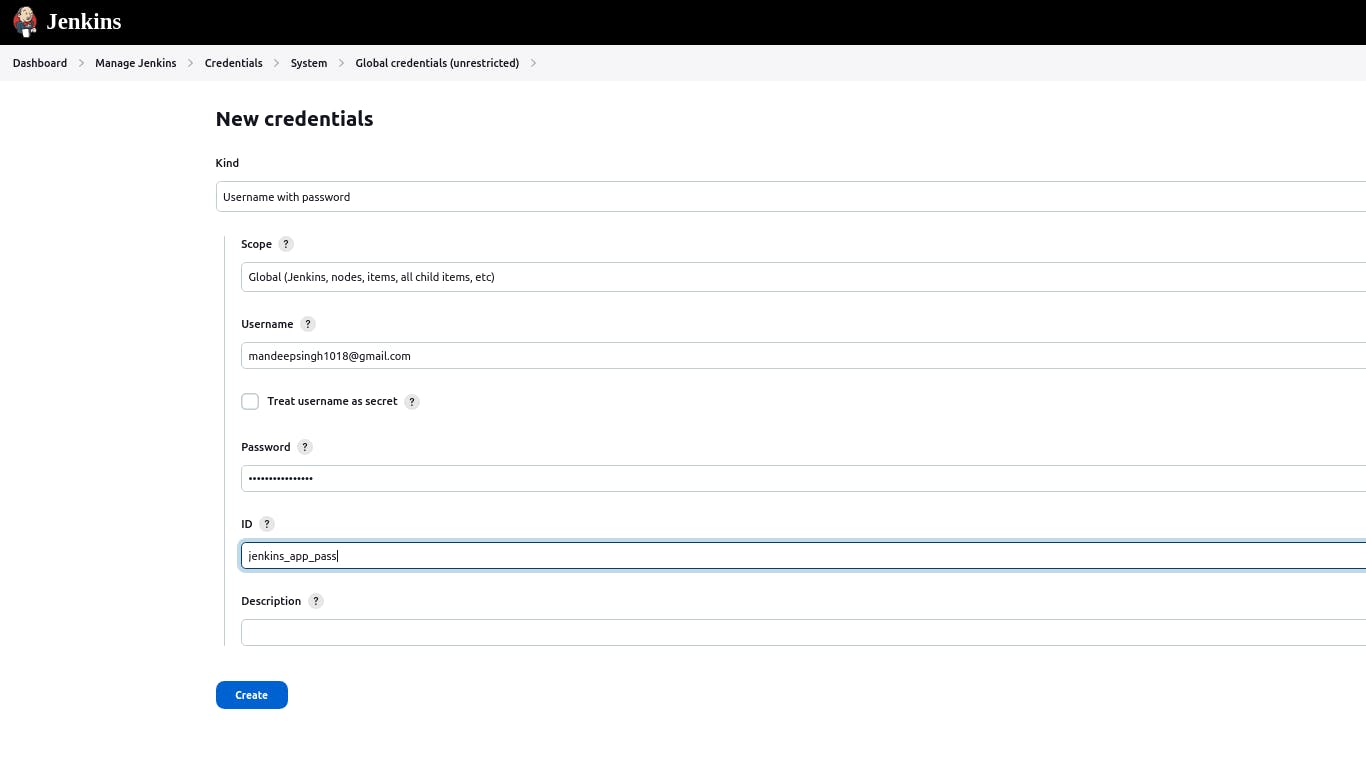
* + - * + Go to your Google Account.
        + Select **Security**
        + Under "**Signing in to Google**," select **2-Step Verification**.
        + At the bottom of the page, select **App passwords**.
        + Enter a name that helps you remember where you’ll use the app password.
        + Select **Generate**.
        + To enter the app password, follow the instructions on your screen. The app password is the 16-character code that generates on your device.
        + Select **Done**.
      1. Use the created app password in the Email configuration for Jenkins (Step 2) After filling the required fields, it will look similar to this. Click on Test Configuration.



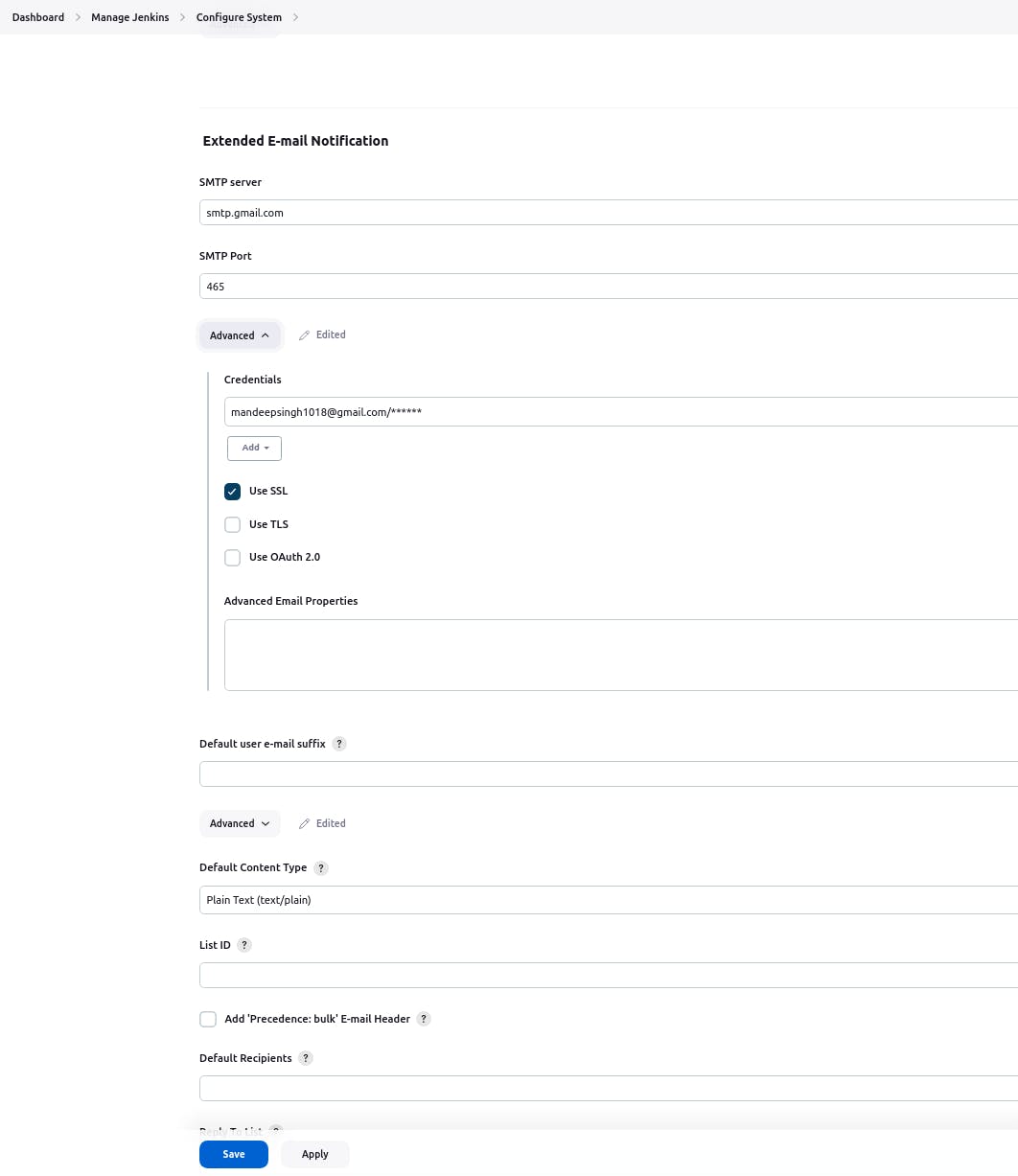
* + - 1. Our test config is working properly, let's setup the email now.

First, we have to create a credential to use the app password in the jenkins pipeline to allow access jenkins to send emails.

* + - * + Goto Manage Jenkins > Credentials > select global, select kind as Username with password.



* + - * + Goto Manage Jenkins > Configure Systems > Extended E-mail Notification.



* + - * + Add a post block to the jenkins pipeline after the stages block.

COPY

post {

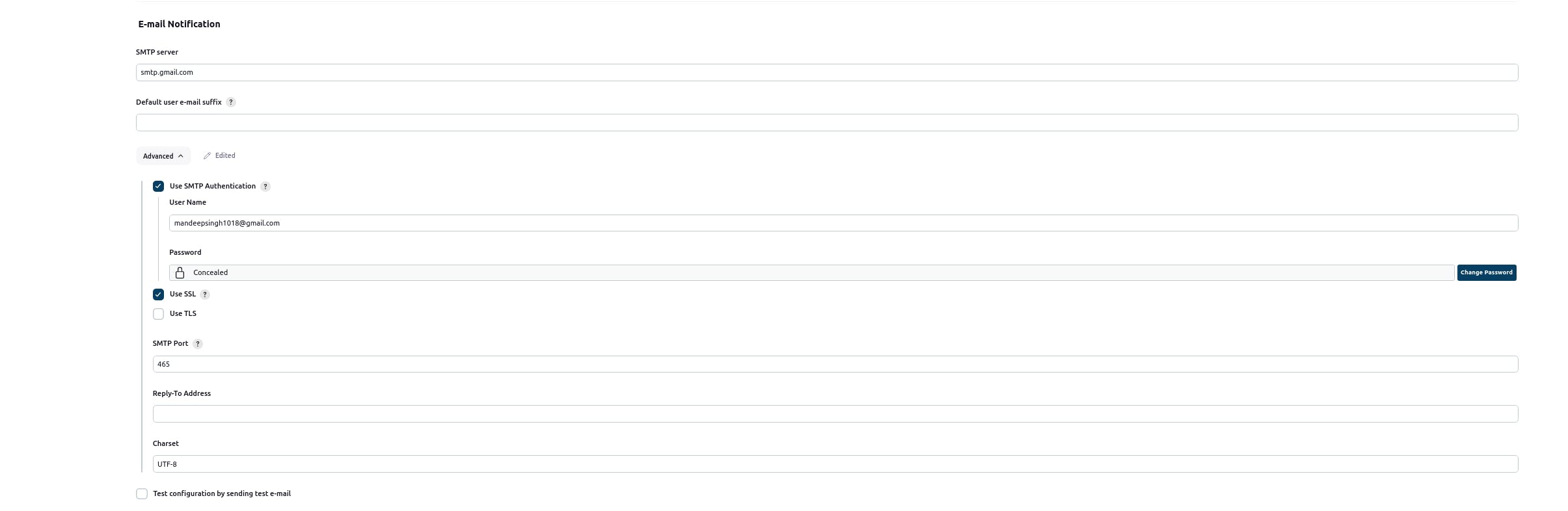
always {

mail bcc: '', body: "<br>Project: ${env.JOB\_NAME} <br>Build Number: ${env.BUILD\_NUMBER} <br>Build URL: ${env.BUILD\_URL}", cc: '', charset: 'UTF-8', from: '', mimeType: 'text/html', replyTo: '', subject: "${currentBuild.result} CI: Project name -> ${env.JOB\_NAME}", to: "mandeepsingh1018@gmail.com";

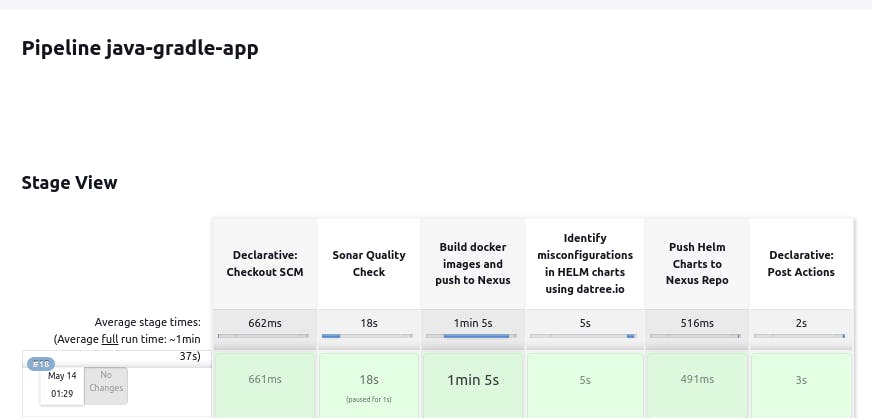
}

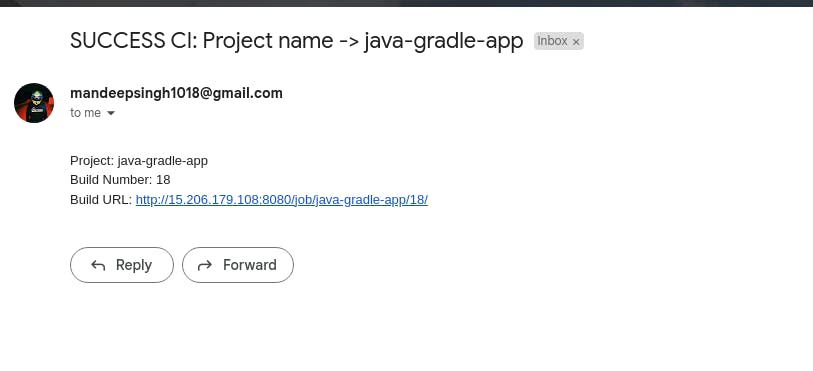
}

* + - * + Also make sure that you have filled the Email Notification section as well in addition to the Extended Email Notification, if this is missed then the build will fail.

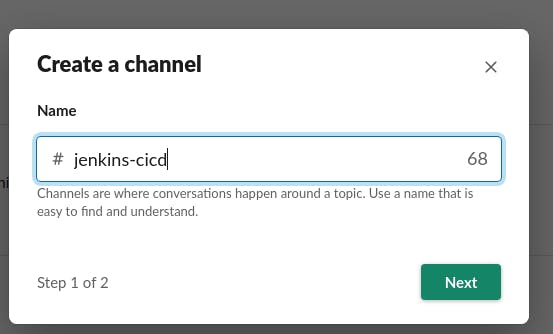


* + - * + Now build the job again, this time you'll recieve an email about the Status of the job.

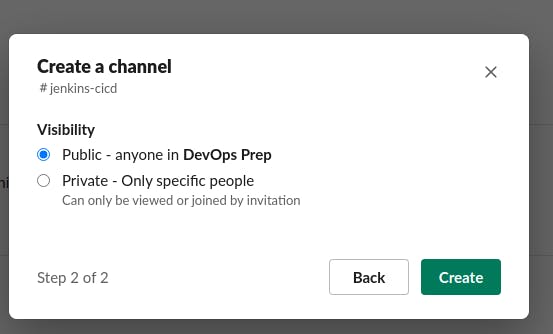




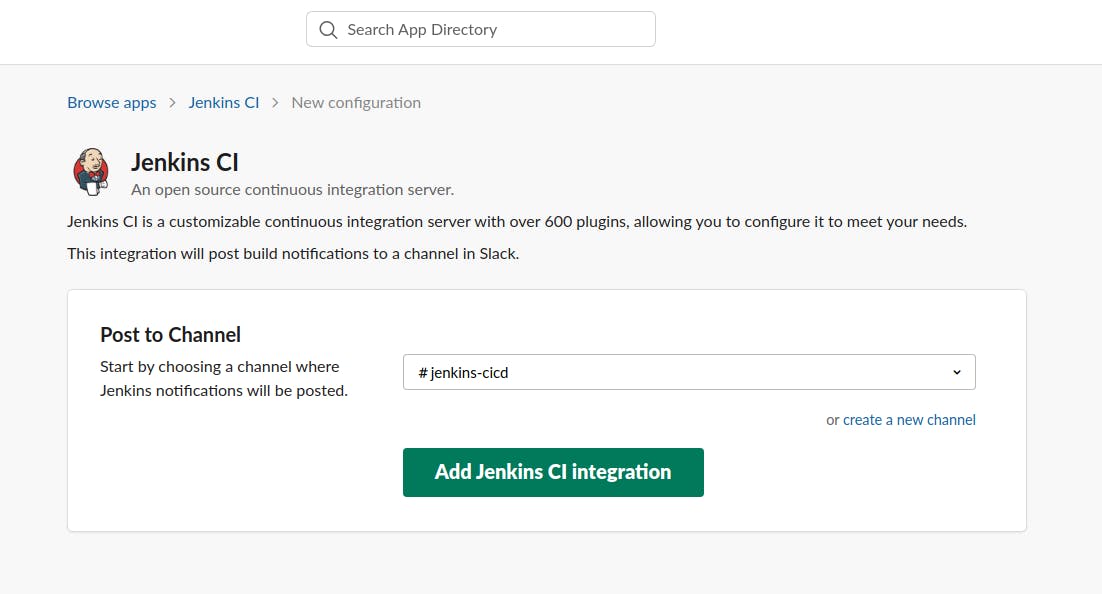
* + - **Configure Slack notifications**
      1. Download Slack plugin
      2. Goto Manage Jenkins > Manage Plugins > Available, search for slack and download the plugin
      3. Login to your slack account and create a channel for jenkins notifications.



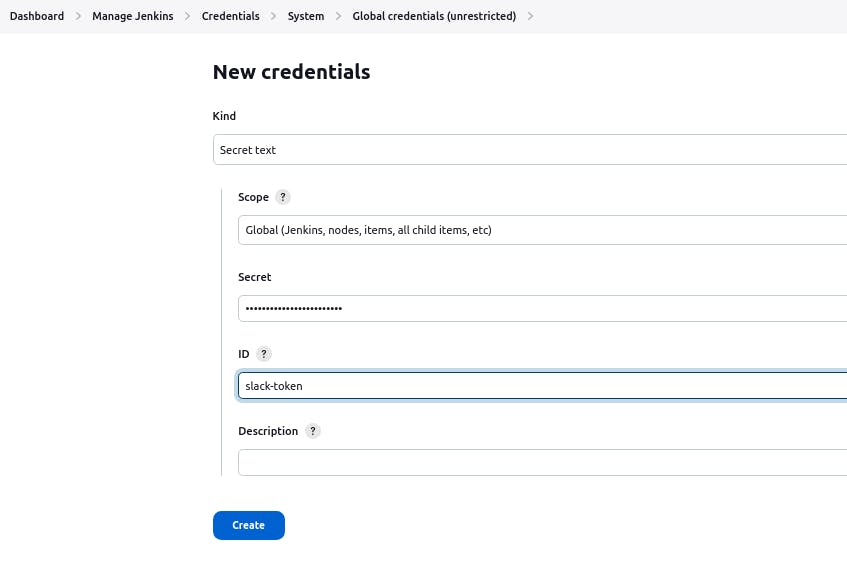
* + - 1. On next screen, select Public and then click on Create.



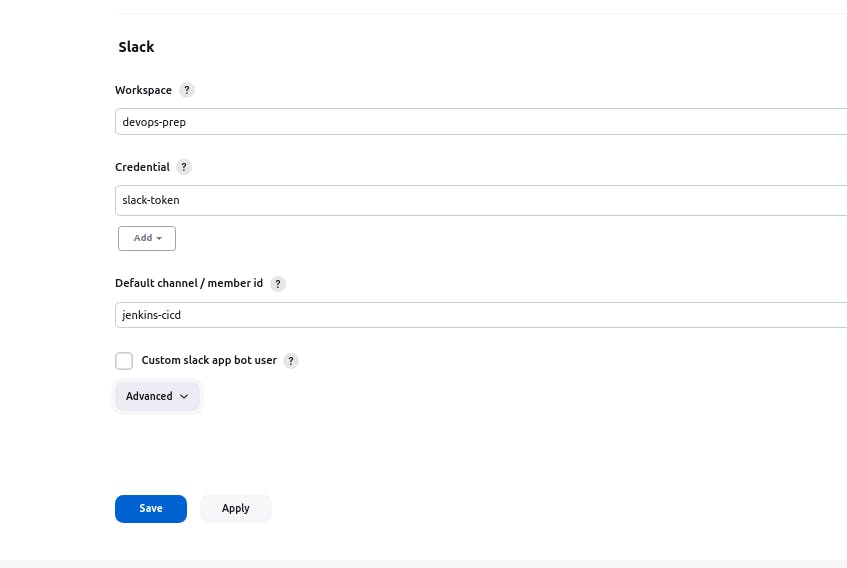
* + - 1. Goto <https://my.slack.com/services/new/jenkins-ci>



* + - 1. Select the newly created slack channel for jenkins and click on Add Jenkins CI Integration.
      2. Follow the on-screen instructions in step4 and get the **workspace (team subdomain)**, and **Integration Token Credential ID**
      3. Create a secret text credential with **Integration Token Credential ID**



* + - 1. Goto Manage Jenkins > Configure Systems, Search for Slack settings.



* + - 1. Add this to the post block in your pipeline, you can also choose to change the frequency to always, success, or failure etc, I have added it in the same always block where mail is configured.

slackSend channel: '#jenkins-cicd', message: "Project: ${env.JOB\_NAME} \n Build Number: ${env.BUILD\_NUMBER} \n Status: \*${currentBuild.result}\* \n More info at: ${env.BUILD\_URL}"

COPY

post {

always {

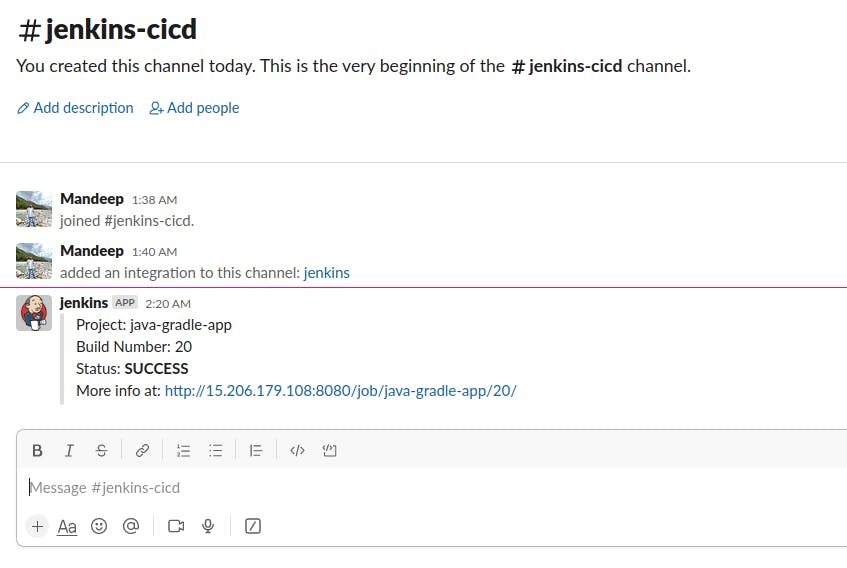
mail bcc: '', body: "<br>Project: ${env.JOB\_NAME} <br>Build Number: ${env.BUILD\_NUMBER} <br>Build URL: ${env.BUILD\_URL}", cc: '', charset: 'UTF-8', from: '', mimeType: 'text/html', replyTo: '', subject: "${currentBuild.result} CI: Project name -> ${env.JOB\_NAME}", to: "mandeepsingh1018@gmail.com";

slackSend channel: '#jenkins-cicd', message: "Project: ${env.JOB\_NAME} \n Build Number: ${env.BUILD\_NUMBER} \n Status: \*${currentBuild.result}\* \n More info at: ${env.BUILD\_URL}"

}

}

* + - 1. Make sure to change the TeamDomain and channel name.
      2. Make the changes to Jenkinsfile and push it to the dev branch.
      3. Now, let's build the job and check if we get any slack notifications.



* + - **Deploy Helm Charts on k8s cluster**
      1. Let's start by checking if we have any release created on our cluster.

COPY

jenkins@jenkins:~$ helm list

NAME NAMESPACE REVISION UPDATED STATUS CHART APP VERSION

We don't have any helm releases

* + - 1. Let's add a new stage to our jenkins pipeline to create a release

COPY

stage('Deploy application on k8s-cluster') {

steps {

script{

dir ("kubernetes/"){

sh 'helm upgrade --install --set image.repository="$DOCKER\_HOSTED\_EP/javawebapp" --set image.tag="${VERSION}" jwa1 myapp/ '

}

}

}

}

* + - * + The steps block contains a script block that changes the working directory to kubernetes/ using the dir directive. This is where the Helm chart and the Kubernetes deployment configuration files are stored.
        + The command is a Helm CLI command that upgrades an existing Kubernetes deployment or installs a new one using a Helm chart located in the myapp/ directory.
        + The command is broken down as follows:

helm upgrade: This command upgrades a Helm release if it already exists, or installs a new one if it does not exist.

--install: This flag indicates that a new release should be installed if it does not already exist.

--set: This flag sets one or more values in the Helm chart's values file. In this case, it sets the image.repository and image.tag values to the specified values.

image.repository="13.235.91.151:8083/javawebapp": This sets the Docker image repository for the application to 13.235.91.151:8083/javawebapp.

image.tag="${VERSION}": This sets the Docker image tag for the application to a value stored in the ${VERSION} environment variable defined in the pipeline

jwa1: This is the name of the Helm release.

myapp/: This is the path to the Helm chart directory containing the values.yaml file and any other necessary files.

* + - 1. Create a a kubernetes secret for nexus docker-hosted repo

COPY

kubectl create secret docker-registry registry-secret --docker-server=13.235.91.151:8083 --docker-username=admin --docker-password=msx@9797 --docker-email=not-needed@yolo.com

* + - * + We need this secret to authenticate to the nexus repo hosted on our nexus server
        + This is a kubectl command that creates a Kubernetes secret named registry-secret of type docker-registry. This secret is used to store credentials for authenticating with a Docker registry located at the specified IP address and port number  
          (13.235.91.151:8083). The --docker-username, --docker-password, and --docker-email flags are used to set the corresponding authentication credentials for the Docker registry.
        + When we run the commands mentioned in **step2**, the image.tag will get the value of $VERSION variable that is build number and it will also be replaced in the values.yaml file in helm charts directory

COPY

# Default values for myapp.

# This is a YAML-formatted file.

# Declare variables to be passed into your templates.

replicaCount: 2

image:

repository: IMAGE\_NAME

### this is image.repository, it will be replaced by nexus\_ip:8083/springapp

pullPolicy: IfNotPresent

# Overrides the image tag whose default is the chart appVersion.

tag: IMAGE\_TAG

#### this is image.tag, it will be replaced by build number of the pipeline

service:

type: NodePort

port: 8080

* + - * + Now when the helm charts command helm upgrade --install --set image.repository="13.235.91.151:8083/javawebapp" --set image.tag="${VERSION}" jwa myapp/ will be excuted it will replace all the values of variables, labels, annotation from the helm helper, chart.yaml and values.yaml in the deployment.yaml and services.yaml
        + In deployment.yaml, in the pod template section, we have a field called imagePullSecrets, this field indicates that the container images for the pods are stored in a private repository.

COPY

template:

metadata:

labels:

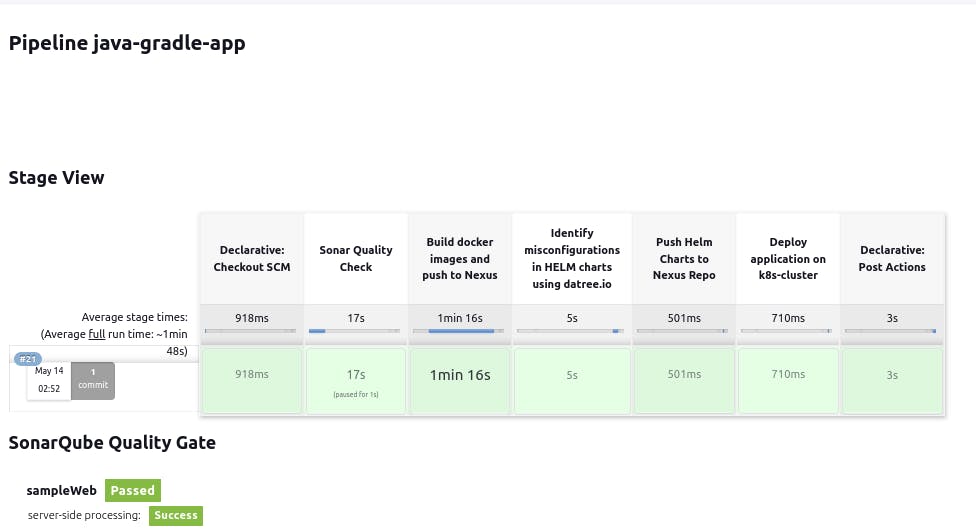
{{- include "myapp.selectorLabels" . | nindent 8 }}

spec:

imagePullSecrets:

- name: registry-secret

* + - * + The secret registry-secret is the same secret that we created.
        + Commit the Jenkinsfile changes and start a build.



* + - * + Build was successful, let's verify the application deployment on the k8s cluster.

COPY

jenkins@jenkins:~$ kubectl get nodes

NAME STATUS ROLES AGE VERSION

k8s-master Ready control-plane 12h v1.27.1

k8s-node1 Ready <none> 12h v1.27.1

jenkins@jenkins:~$ kubectl get deployments

NAME READY UP-TO-DATE AVAILABLE AGE

jwa1-myapp 2/2 2 2 3m59s

jenkins@jenkins:~$ kubectl get pods

NAME READY STATUS RESTARTS AGE

jwa1-myapp-67cd67cbb8-n2w4x 1/1 Running 0 4m5s

jwa1-myapp-67cd67cbb8-s895p 1/1 Running 0 4m5s

jenkins@jenkins:~$ kubectl get svc

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

jwa1-myapp NodePort 10.100.153.6 <none> 8080:30984/TCP 4m10s

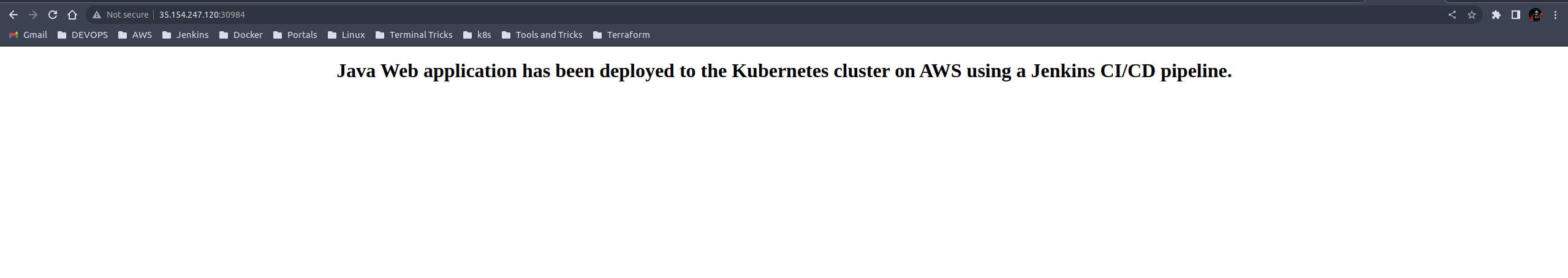
kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 12h

jenkins@jenkins:~$ helm list

NAME NAMESPACE REVISION UPDATED STATUS CHART APP VERSION

jwa1 default 1 2023-05-13 21:24:01.463472615 +0000 UTC deployed myapp-0.2.0 1.16.0

* + - * + The deployment is up and the pods are running, let's try to access the application from the web browser using the node\_ip followed by NodePort, 35.154.247.120:30984



* + - * + The java web application is up and running, we have successfully deployed our application on a k8s cluster using helm charts.
    - **Add approval request to manually approve deployments**
      1. We can use this stage block to add a manual approval to deploy or abort the deployment.

COPY

stage("Deployment Approval"){

steps{

script{

timeout(10){

mail bcc: '', body: "<br>Project: ${env.JOB\_NAME} <br>Build Number: ${env.BUILD\_NUMBER} <br> Goto : ${env.BUILD\_URL} and approve/reject the deployment", cc: '', charset: 'UTF-8', from: '', mimeType: 'text/html', replyTo: '', subject: "CICD APPROVAL REQUEST: Project name -> ${env.JOB\_NAME}", to: "mandeepsingh1018@gmail.com";

slackSend channel: '#jenkins-cicd', message: "\*CICD Approval Request\* \nProject: \*${env.JOB\_NAME}\* \n Build Number: ${env.BUILD\_NUMBER} \n Status: \*${currentBuild.result}\* \n Go to ${env.BUILD\_URL} to approve or reject the deployment request."

input(id: "DeployGate", message: "Approval required to proceed, deploy ${env.JOB\_NAME}?", ok: 'Deploy')

}

}

}

}

COPY

input(id: "DeployGate", message: "Approval required to proceed, deploy ${env.JOB\_NAME}?", ok: 'Deploy')

* + - 1. We have use an input block to register input from the approvers.
      2. Let's push the code and build the job
      3. We will receive email and slack notifications like this:

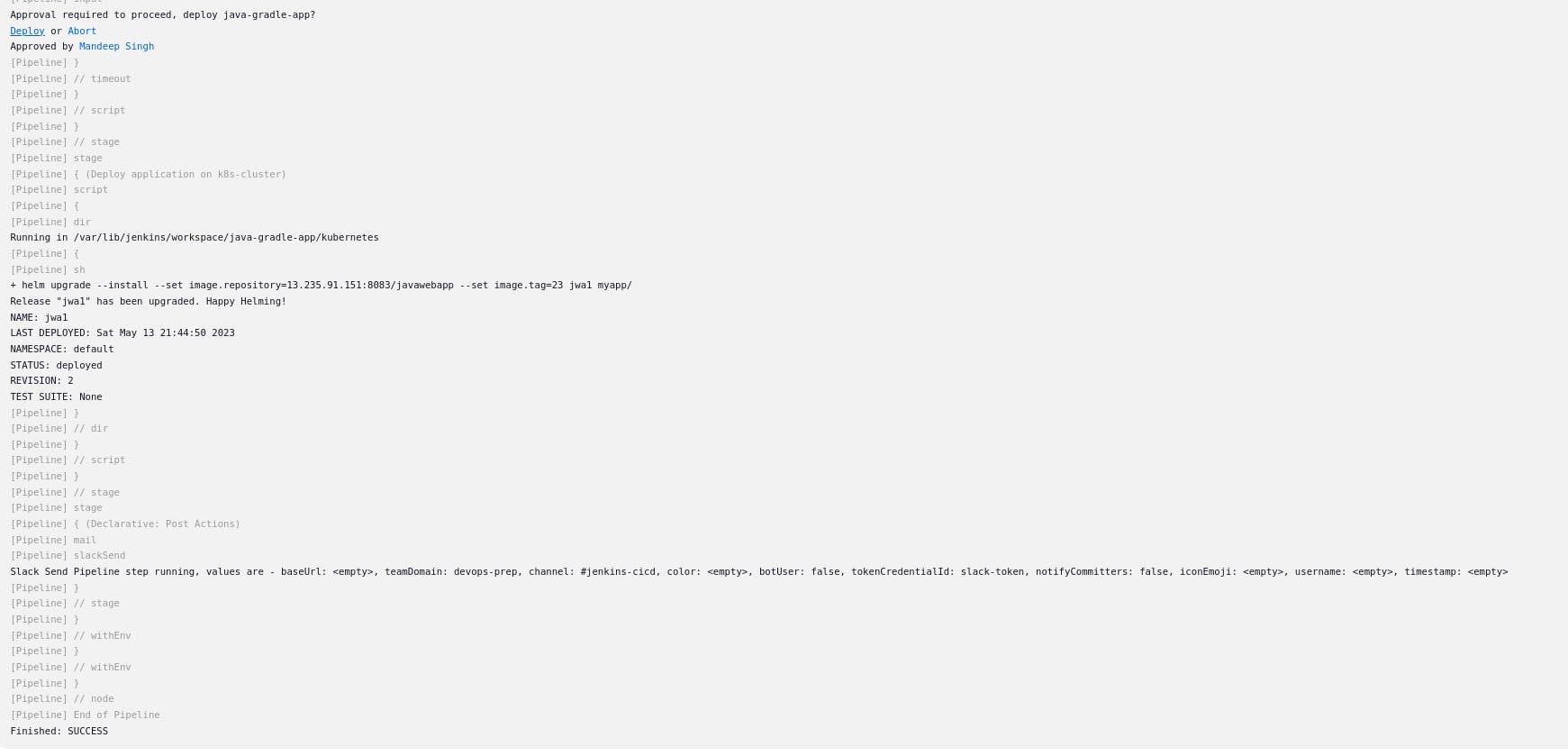




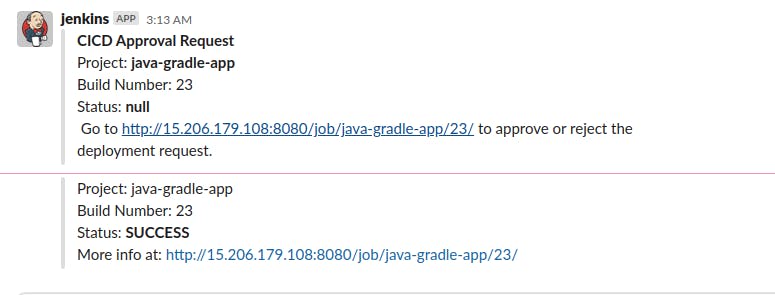
* + - 1. We can use the build URL mentioned in the notifications to approve or reject the deployment request.



* + - 1. Jenkins was waiting for us to manually Deploy or Abort the deployment request. Once we approved the request, the application was deployed to the k8s cluster.



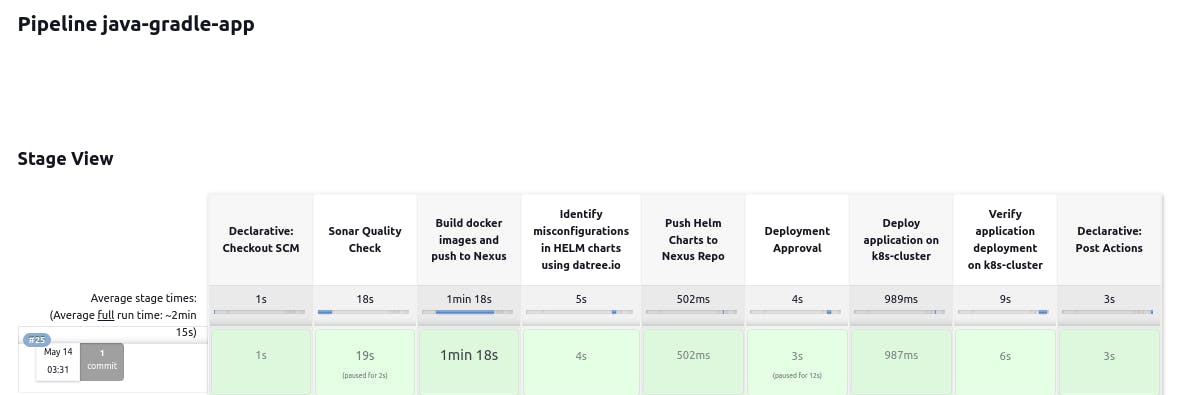
* + - 1. Received slack notification for the same



* + - 1. First notification is for approving the deployment request and the second one is for the Build status update.
      2. This concludes Stage V of our pipeline.

**Stage VI: Verify application deployment on k8s cluster**

* COPY
* kubectl run curl --image=curlimages/curl -i --rm --restart=Never -- curl myjavaapp-myapp:8080
* We can verify the application deployent using the kubectl command. We create a pod with curl image which curls the service at 8080 port and then the pod is deleted after it's work is done.
* If the exit code of this commad is 0 that means our application deployment was a success else it was a failure
* COPY
* stage("Verify application deployment on k8s-cluster") {
* steps {
* script{
* dir ("kubernetes/"){
* sh 'kubectl run curl --image=curlimages/curl -i --rm --restart=Never -- curl jwa1-myapp:8080 '
* }
* }
* }
* }
* Add this stage to the Jenkinsfile, push the code to dev branch and start the build.



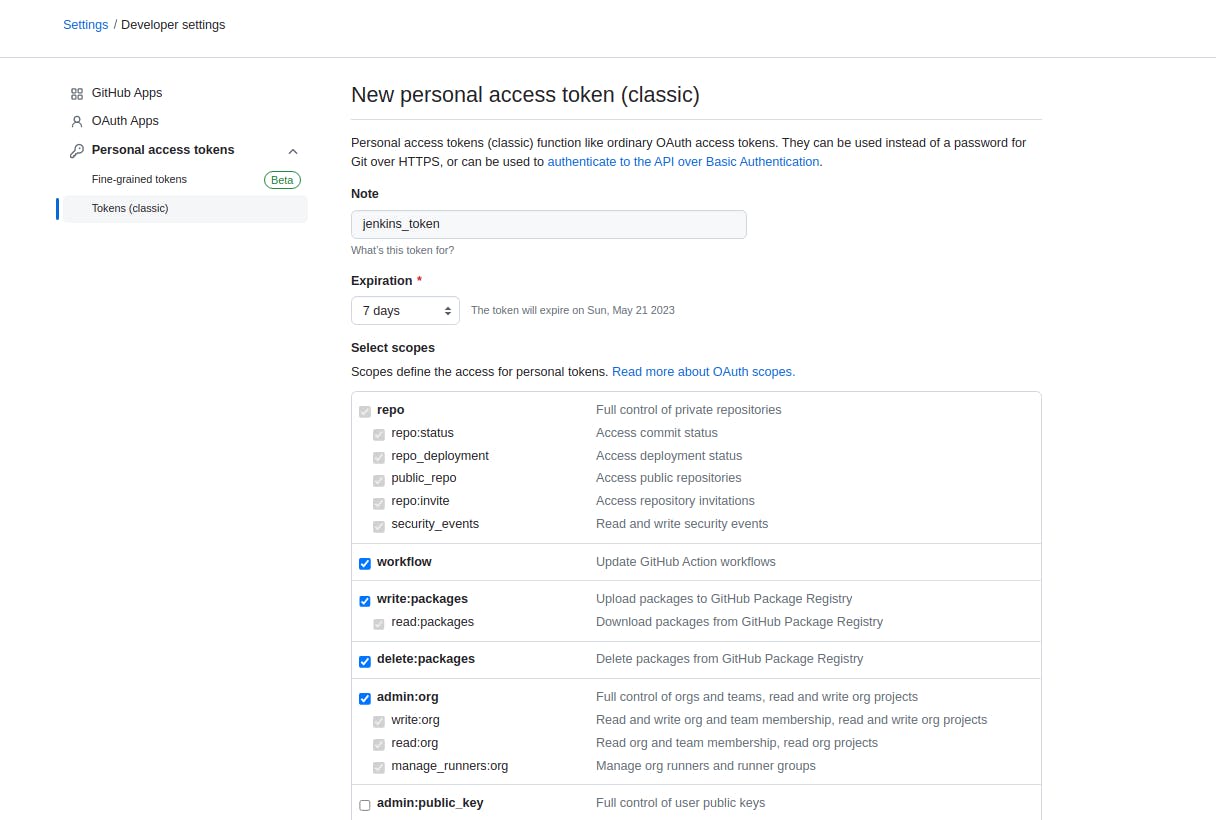
* All the stages were successful. With this all the Stages of our application are now completed.
* One last thing is left, that is enabling pull request triggers, let's do that.

**Configure PR based trigger in Jenkins**

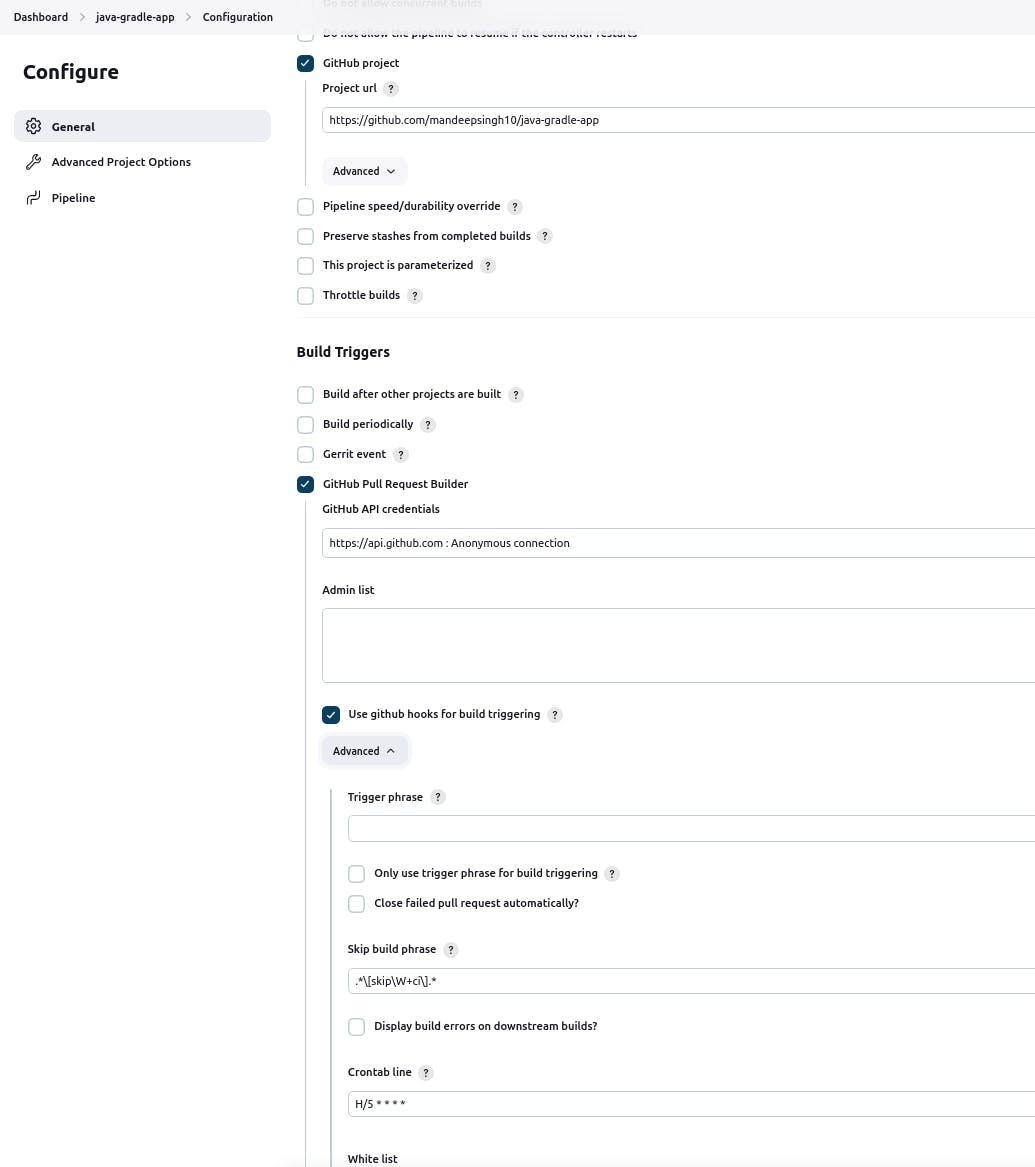
* We should never make changes to the **main/master** branch of the project, we should create separate branches, develop the code and then we should create a Pull request to merge the changes of our request into the **main** branch.
* It is a common practice that when we raise a pull request it will automatically trigger the CICD Pipeline and then update back the status of the CI pipeline onto the pull request, because reviewers should also check who has made what changes and that the changes should not break our CICD pipeline.
* Let's setup PR based Triggers, install the **GitHub Pull Request Builder** plugin.
* Navigate to Manage Jenkins > Configure System, search for **GitHub Pull Request Builder** section.
* In this section we need to add our github creds, so go to your github account and navigate to Settings > Developer settings > Personal access tokens

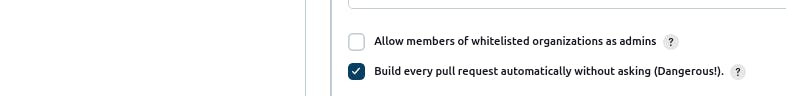
Select Tokens (classic), click on Generate new token  
then Generate new token (classic). You will be prompted to enter your github password, enter the password and proceed.

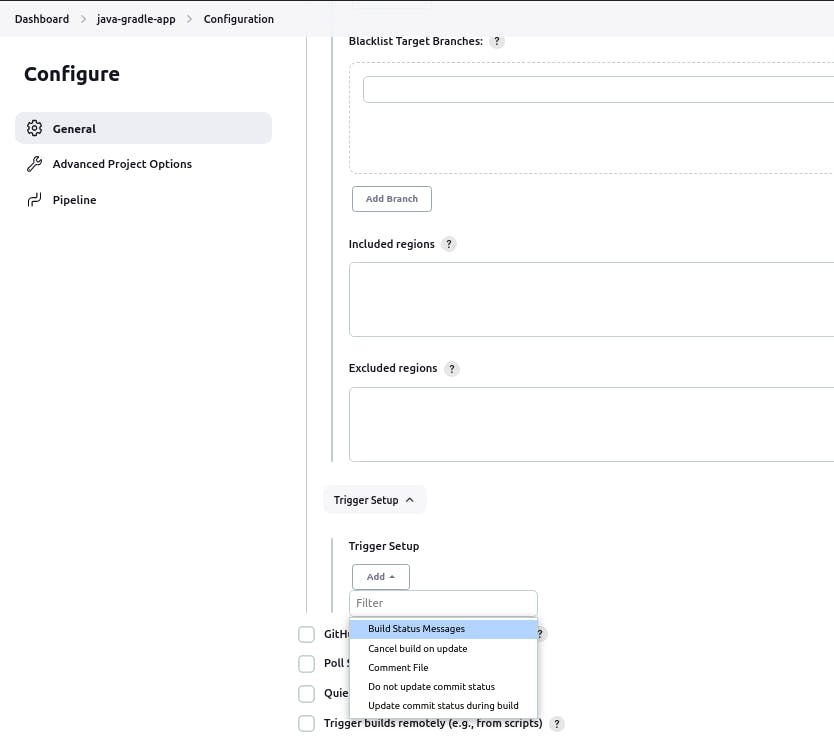
* Give a name to the Token, choose expiration and then select relevant scopes.

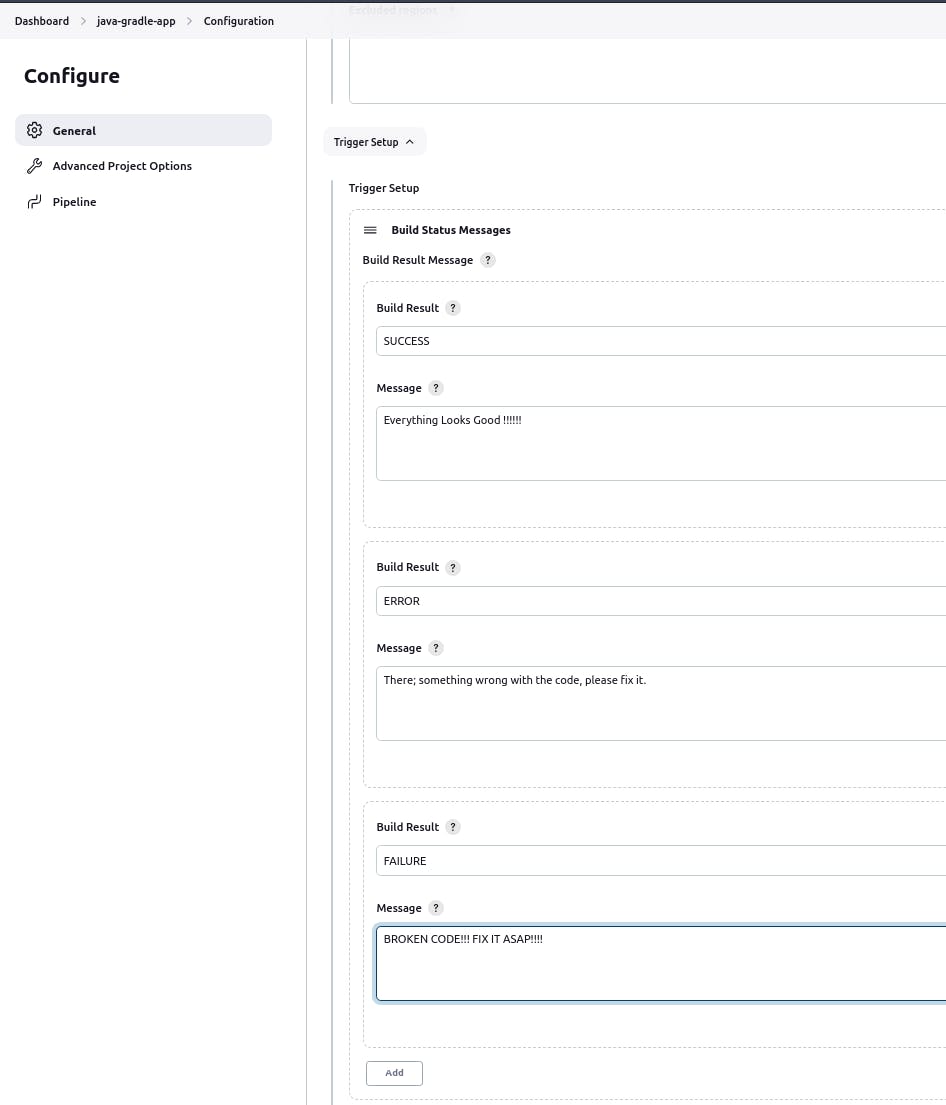


* Once done, click on **Generate Token.**
* Create a Jenkins credential of Kind Username and Password using this token as password and your github account name as username.
* Now, navigate to the project and go to configuration and under general, select the following things.
  1. Add a project URL
  2. Tick the option - GitHub Pull Request Builde, also do the same for  
     Use github hooks for build triggering.
  3. Build every pull request automatically without asking (Dangerous!).' This whitelist everyone, and it's not advisable in an actual production setting but as we are doing a project for learning we can enable it. Otherwise, it's better to add an organization or add people individually, and only yourself as the Admin.



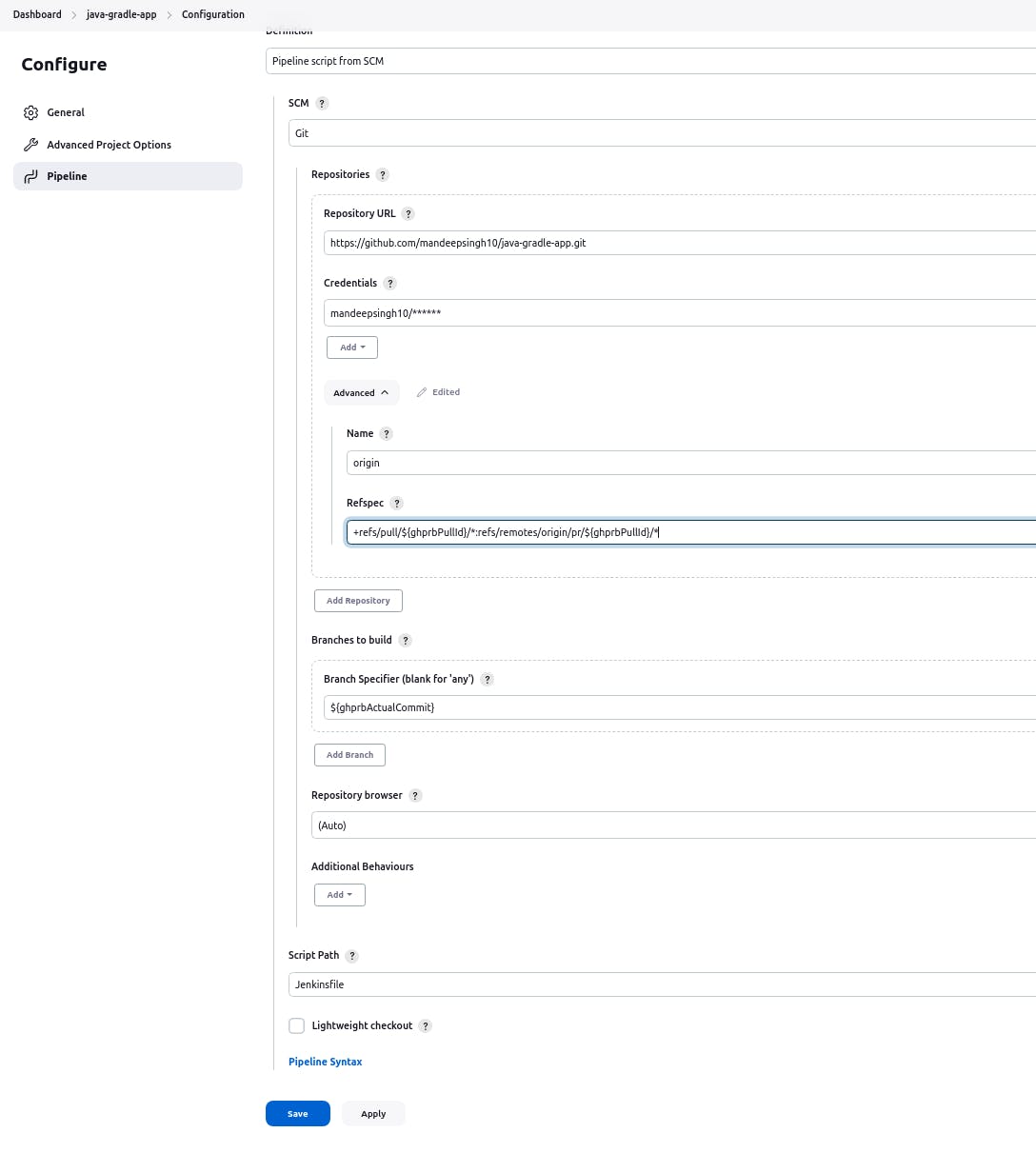


* Now go to Trigger setups and Add Build Status Messages
* 

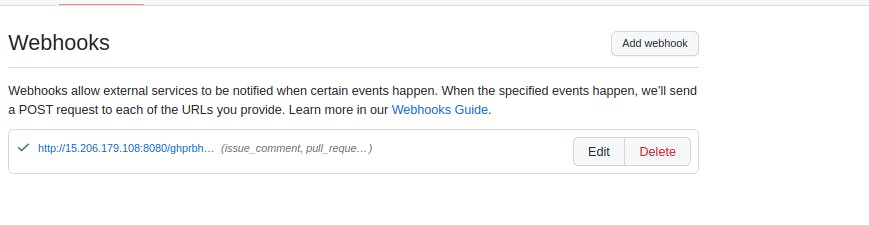


Now go to the Pipeline section and provide the credentials, same one that we just created using PAT and few other settings.

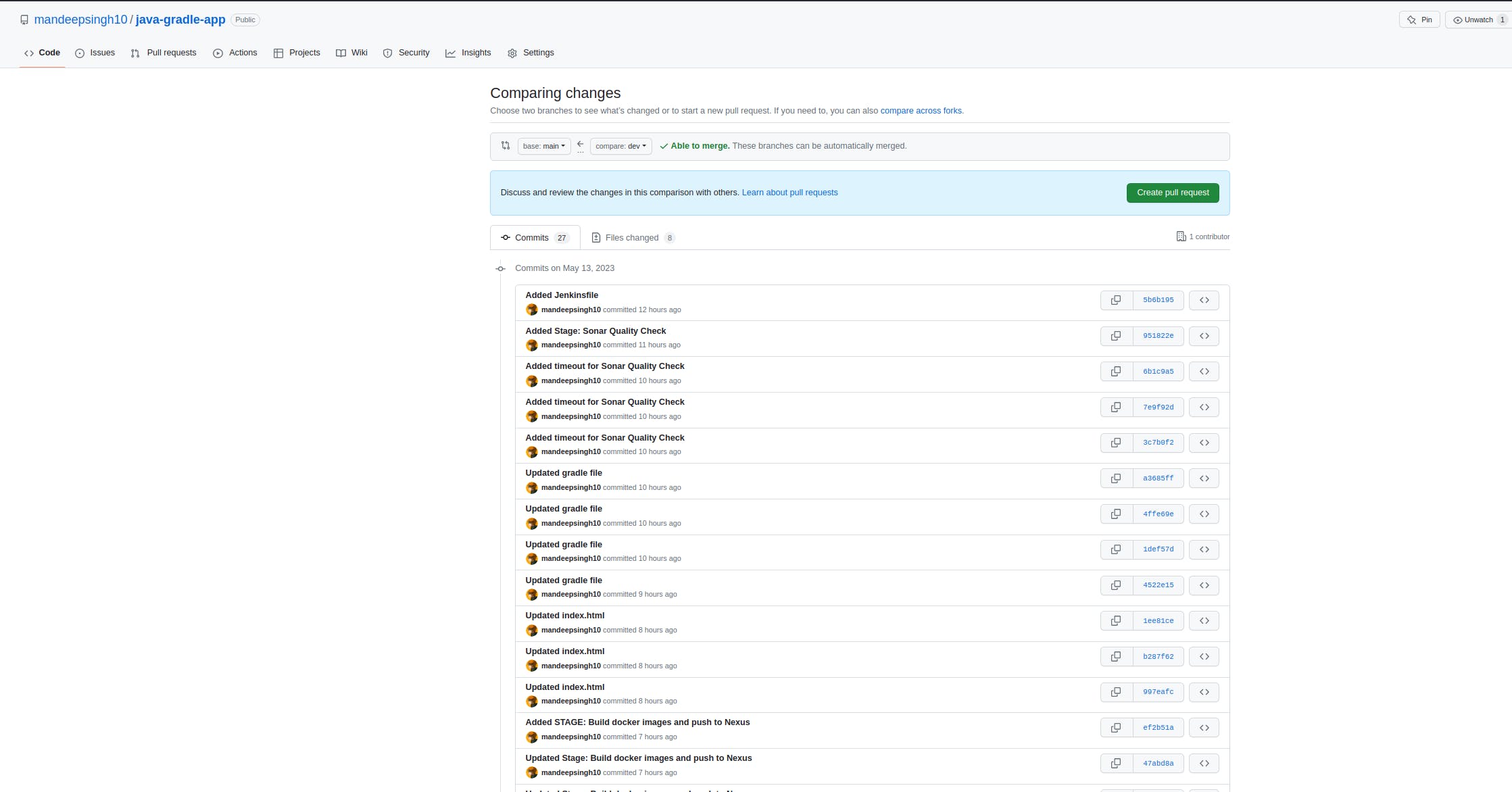
* Update the Refspec and Branch Specifier fields. +refs/pull/${ghprbPullId}/\*:refs/remotes/origin/pr/${ghprbPullId}/\*  
  By adding a refspec and a branch specifier, including variables that are set by the plugin, the job will always build what is specified by the webhook trigger, that comes from GitHub itself.
* Branch Specifier (blank for 'any') is : ${ghprbActualCommit}



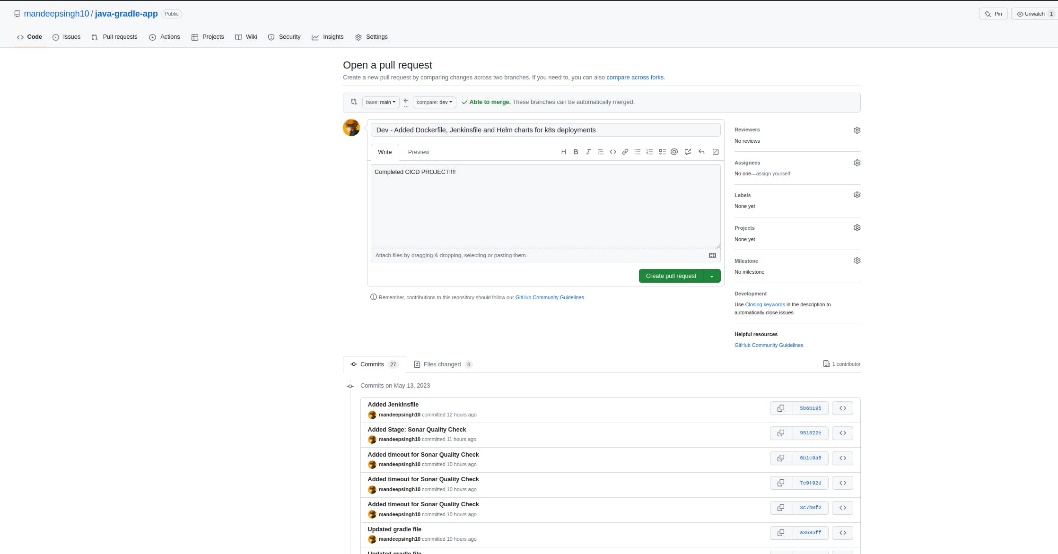
* Next, go back to your github account, then go to the repo settings and then add a webhook http://jenkins\_public\_ip:8080/ghprbhook/
* Select, Let me select individual events, and then select issue comments, Pushes, Pull Requests.
* Click on Add Webhook. After some time there should be a green tick beside the webhook.



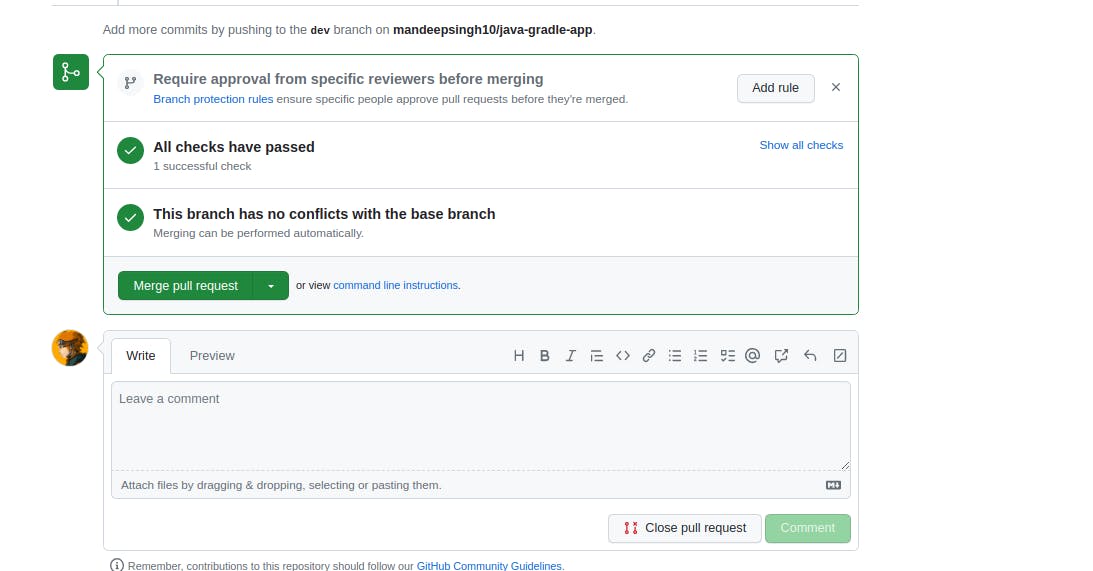
* Now, let's raise a pull request.



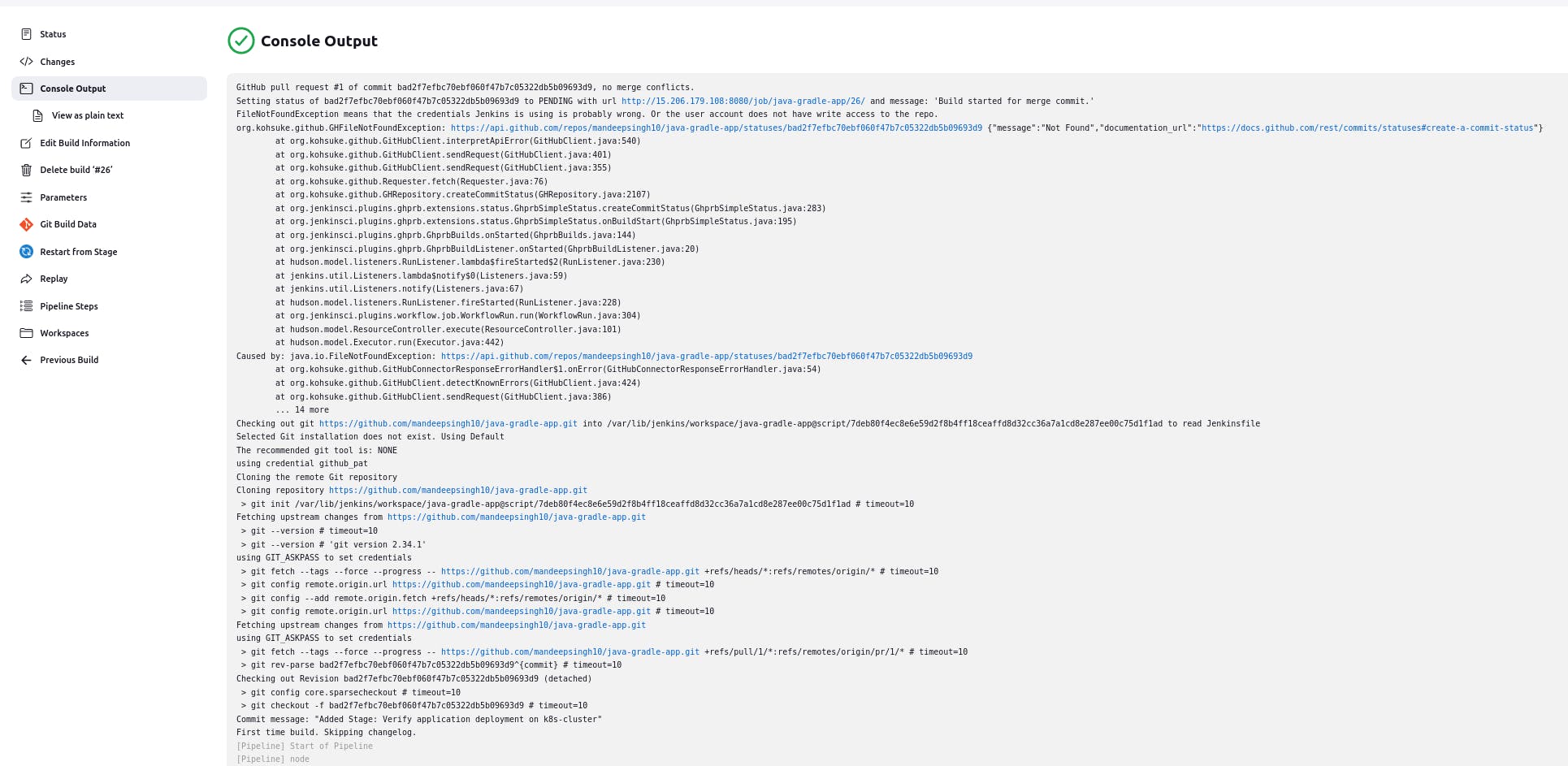
* Click on Create Pull Request.



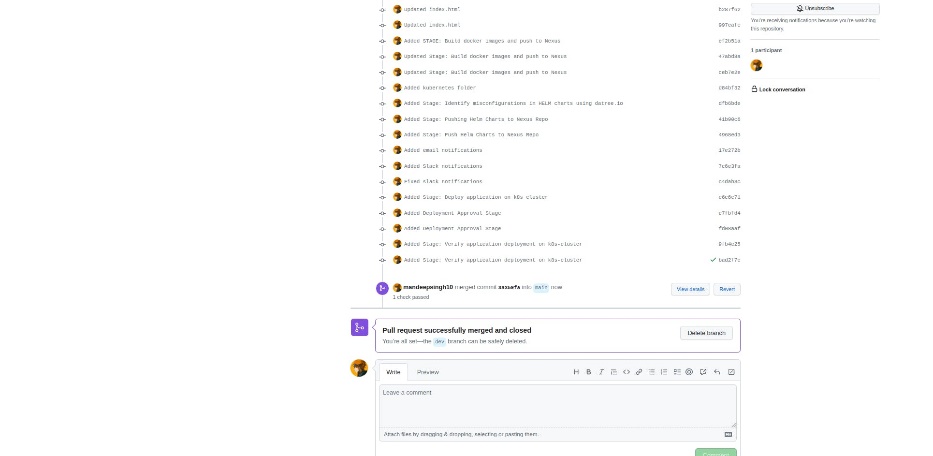
* Next we can add reviewers, but in our case we'll be the only contributors so that's why we won't be able to add a reviewer.
* Next click on **Create Pull Request**. As soon as we create a pull request a trigger has happened.

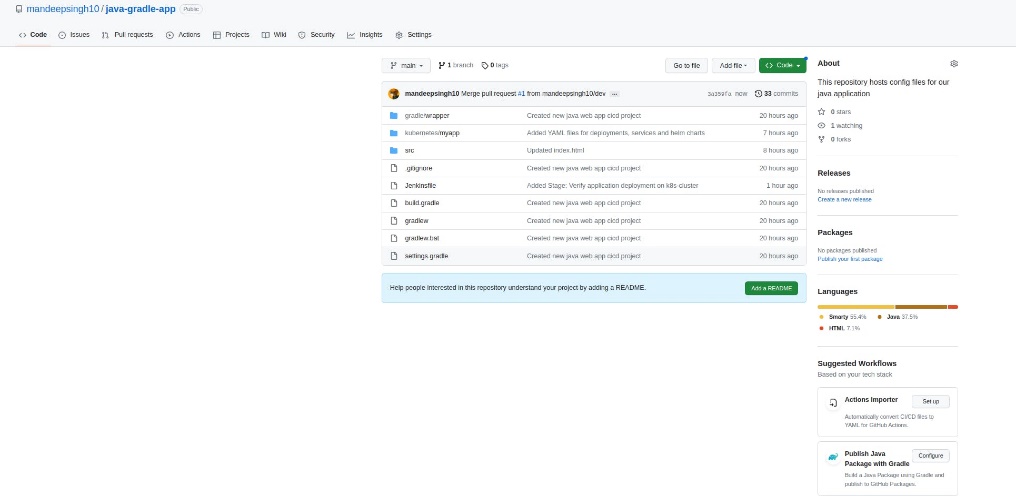


* A jenkins build was triggered.



* Our build was successful so now we can go ahead and merge the pull request.



* Now our dev branch was merged into Main branch, so we don't need the dev branch, we can safely delete it.
* Now we can see that all the updated config files from the dev branch are available in the Main branch.
* 
* We have successfully replicated an advanced end-to-end DevOps pipeline.