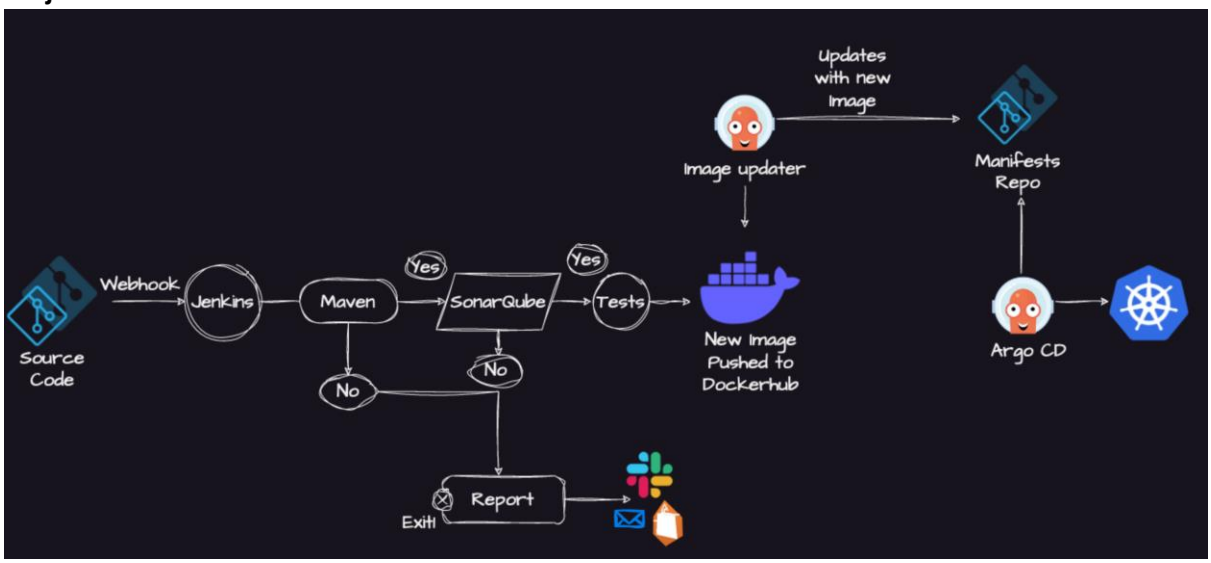
Day 43:

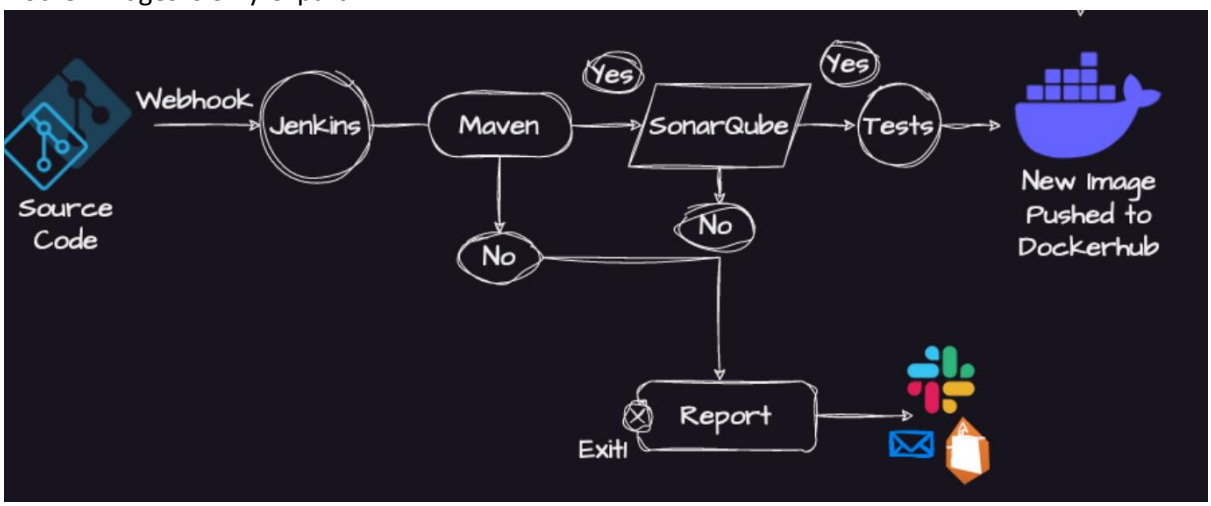
Jenkins Ultimate CICD Pipeline:

Repo: https://github.com/iam-veeramalla/Jenkins-Zero-To-Hero/blob/main/java-maven-sonar-argocd-helm-k8s/README.md

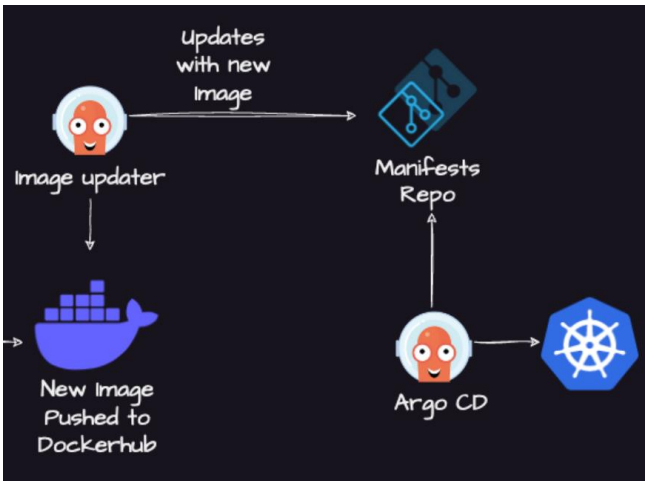
Project Architecture:



We are using two GitHub repos in this project. One for CI and one for CD. First one, with Jenkins till Docker Images is only CI part.



This one is CD Part:



Developer had written Java Code and it is hosted in a Repository. Now Jenkins will watch for the PRs on this repository.

How Jenkins will be notified when any Dev commits any PR? **Using Webhooks.** Jenkins will not watch the repository; GitHub will notify Jenkins. We have to configure the webhook in GitHub. In the GitHub settings we need to define for which actions (Commit, PR, issue) the Webhook needs to be triggered. Whenever developer creates a PR, GitHub will notify Jenkins using Webhooks and ask Jenkins to trigger Pipeline. Then Jenkins starts triggering the Pipeline.

Using Jenkins file we will perform certain actions on Jenkins.

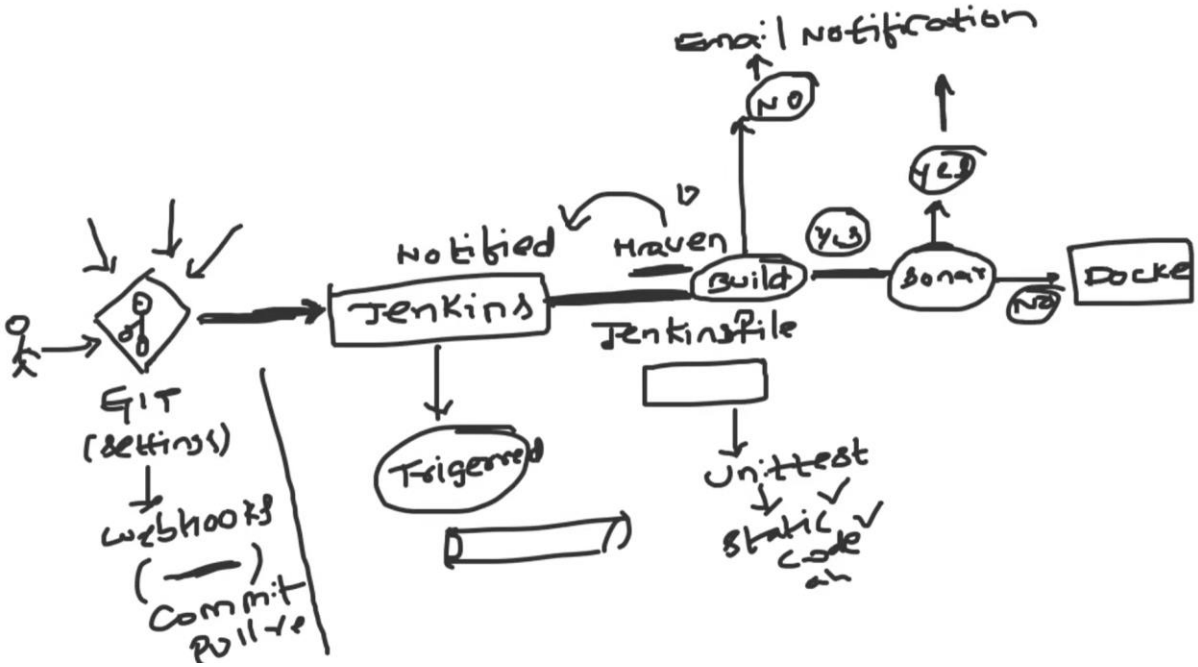
Actions like: Use Maven for building application, during this build stage it will run some Unit tests, If they are successful, we can perform static code analysis;

If build fails, we can configure some alerts using Jenkins plugins. Alerts like email notification, Slack notification using API or if we are on AWS then AWS email notifications.

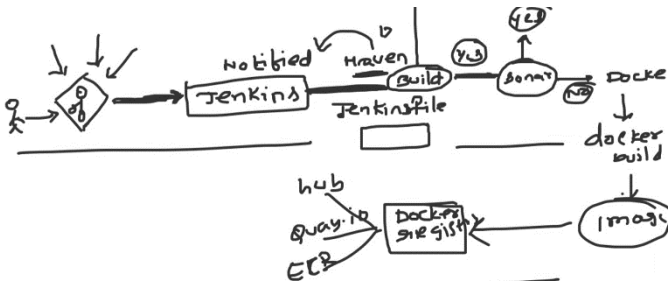
If build passes, we can move to the next stage for checking the security vulnerabilities inside the code or how many errors it had, whether it had reached certain threshold value of the organisation expectations using SonarQube.

If there is security vulnerability then we need to send email notification.

If there is no security vulnerability then we can proceed with the creation of Docker Image.



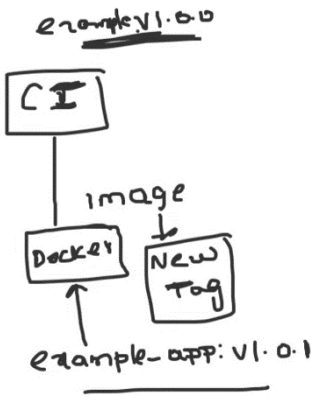
We can add a docker plugin or we can write a shell script to run docker commands or we can use Docker agent to create Images. Once Images are created we will send this image to Docker registry like DockerHub, quay.io or Elastic Container Registry or anything.



This is entire CI Process.

Always prefer Declarative Pipelines which is very easy to write and understand.

Question: What kind of Agent you are using in your pipeline.



Answer: If you are using Docker Agent, we no need to lot of installation.

At the end of CI, we will have a new Artifact (image) and it will have a new tag. We will push this to our Docker Registry. This is the last stage of CI.

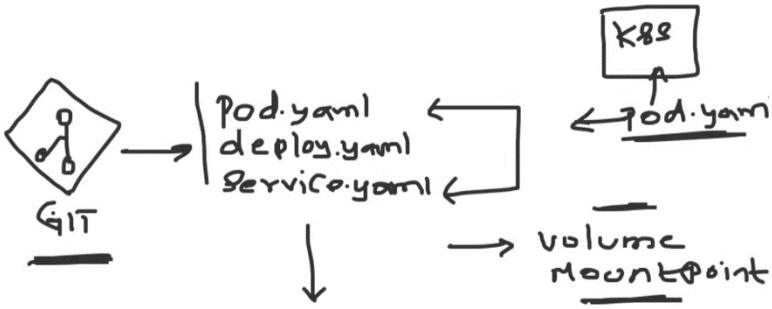
How Does CI and CD communicate with each other?

Earlier they used Ansible Playbooks or Shell scripts after CI Stages in the same pipeline which will push or deploy the artifacts to the target platform.

But Ansible and Shell Scripts are not designed for Continuous Delivery. So, we have to choose a continuous Delivery tools that are based on GitOps. Why GitOps Based?

We need to maintain a separate repo for manifests files like pod/deployment/service YAML. Why?

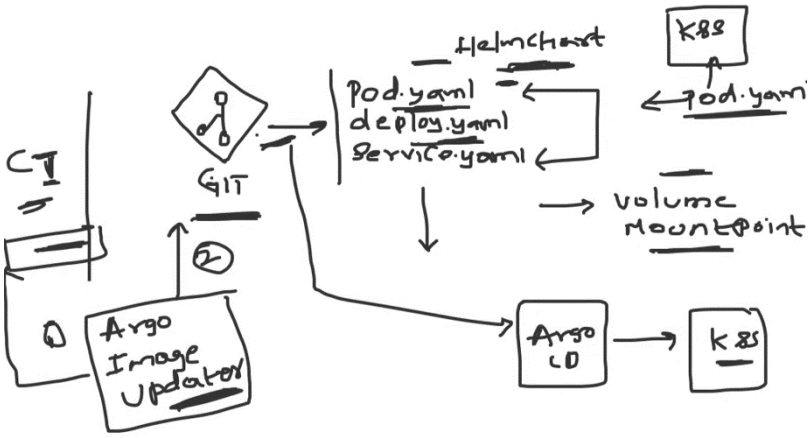
If we are not maintaining separate repo and using same CI pipeline for deployments also then if any changes need to be done in any manifest file or we have to add a volume or mount point to the cluster then we have to login to the K8s cluster and do the change like (pod YAML update) inside the cluster. In this case there is no auditing or code review. CI pipeline will be triggered if there is any PR is created but in this case the deployment part is not having any review.



We have a new image version after CI. How will pod.yml or helm chart should be updated with new version that is created in CI? For this there is very good tool called Argo Image Updater. This will notify and update the GIT repository with new version in th epod.yml/deployment.yml or helm chart where ever it is required.

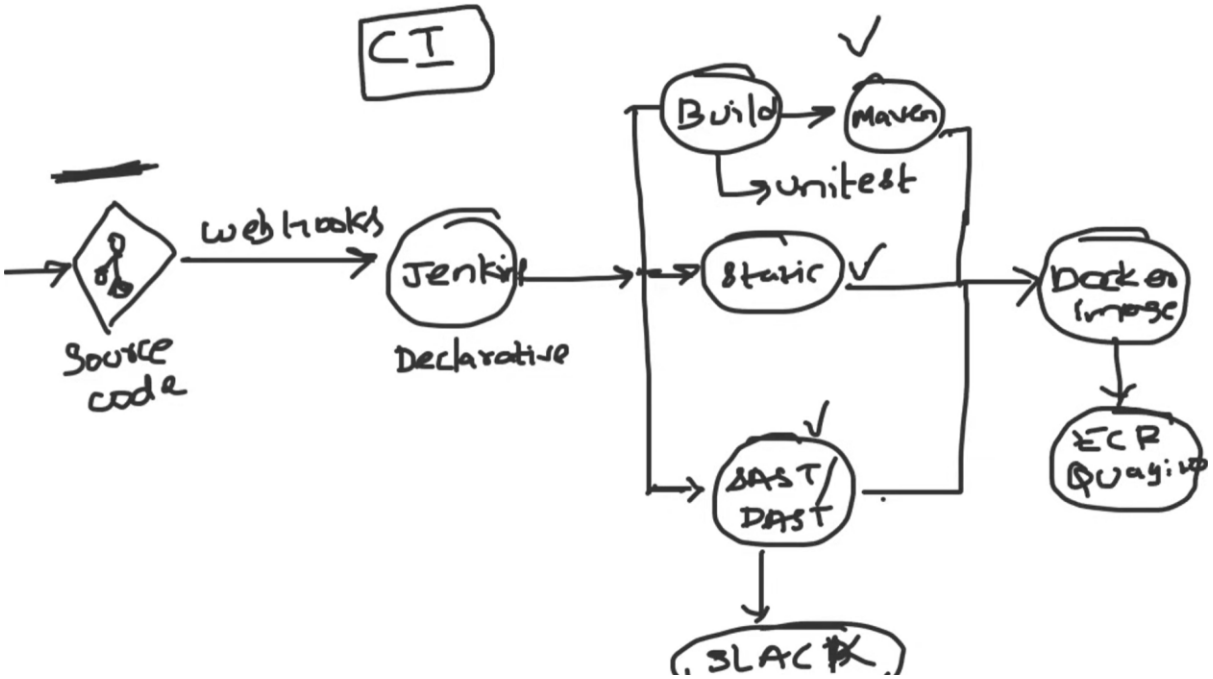
So, Argo Image Updater will monitor the container registry and update the GitHub repository.

Then there is Argo CD which is continuously watching this GitHub repository and takes the new pod.yml / deployment.yml/ helm chart and deploy it to the K8s cluster.

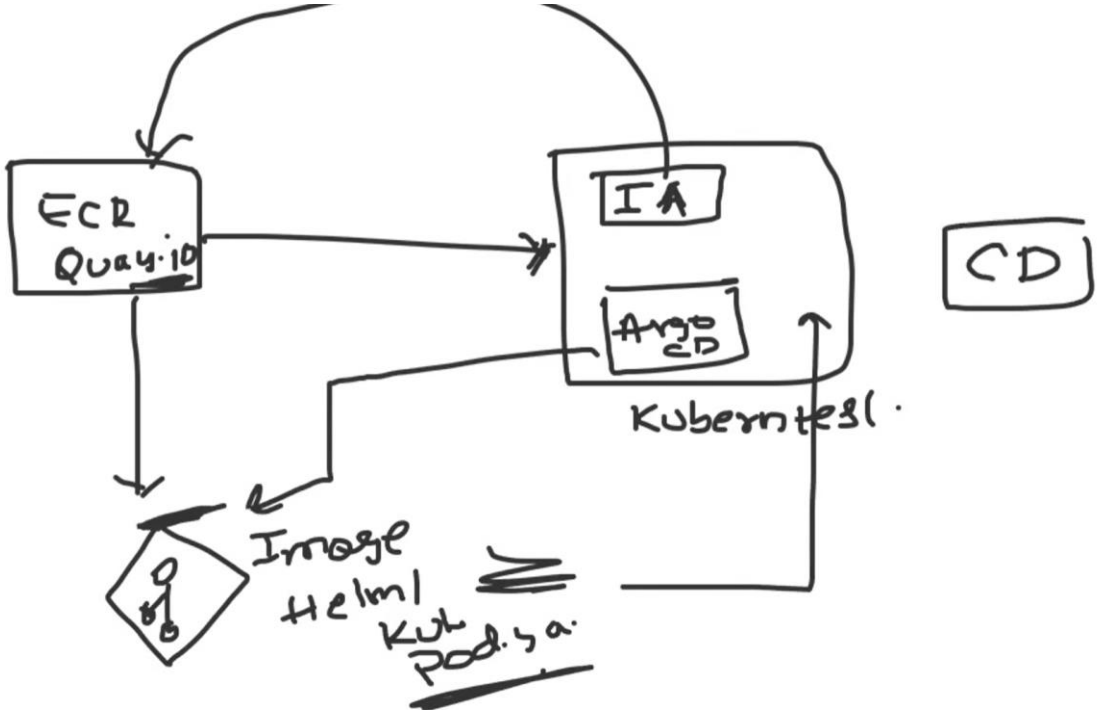


Argo CD will be inside the K8s cluster. It is a K8s controller which always tries to maintain state between the Git repository and Kubernetes cluster. Any change in Manifest files Argo CD will pick and make changes in K8s cluster.

Project Explanation:



We have our source code in Github repo and any change in the repo will be identified by Webhooks and it will trigger jenkins Pipeline which we written in declarative format as it is easy to write and collaborate. In the CI pipleine we have certain actions like we will build the application first where unit test will also happen during build stage and once it is success we go for static code analysis using sonarqube and then we will have SAS/DAS tools to check for any security vulnerabilities and if it failing in any stage we will send slack or email notifications. If nothing is failing any where we will create a docker image and then push it to container registry.



We have Argo Image Updater and Argo Cd inside the K8s cluster as controllers and the Argo Image Updater will continuously monitor the change in container registry and then helps to update the new version in the manifest files or Helm charts in a GitHub repo that is specifically for maintaining manifets files. As soon as the GitHub repo is updated, the other controller Argo CD will take this new image version and deploys in K8s cluster.

This is how we setup our CICD in our organisation.

Project Lab:

Application: This is a simple Sprint Boot based Java application that can be built using Maven. Sprint Boot dependencies are handled using the pom.xml at the root directory of the repository.

This is a MVC architecture based application where controller returns a page with title and message attributes to the view.

Execute the application locally and access it using your browser;

Clone this repo into the PC: https://github.com/VarunTej06/Jenkins-End-to-End

Then move the target directory where the application resides using cd command.

Command: cd Jenkins-End-to-End/spring-boot-app

Pre-Requisites: Maven and Docker

Maven: https://maven.apache.org/install.html

Now, we have to build this application using Maven and then create a Docker image and run it. Then expose the application to localhost and access it.

Follow the step-by-step instructions from here: https://github.com/VarunTej06/Jenkins-End-to-End/tree/main/spring-boot-app

```
→  java-maven-sonar-argocd-helm-k8s git:(main) cd spring-boot-app
→  spring-boot-app git:(main) ls
Dockerfile README.md  pom.xml   src
```

Execute the below commands:

```
→  spring-boot-app git:(main) mvn clean package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------< com.abhishek:spring-boot-demo >-------------------
[INFO] Building spring-boot-demo 1.0
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
```

```
→  spring-boot-app git:(main) ✗ docker build -t ultimate-cicd-pipeline:v1 .

[+] Building 6.5s (9/9) FINISHED
 => [internal] load build definition from Dockerfile                       0.1s
 => => transferring dockerfile: 581B                                       0.0s
 => [internal] load .dockerignore                                          0.0s
 => => transferring context: 2B                                            0.0s
 => [internal] load metadata for docker.io/adoptopenjdk/openjdk11:alpine-jre 5.0s
```

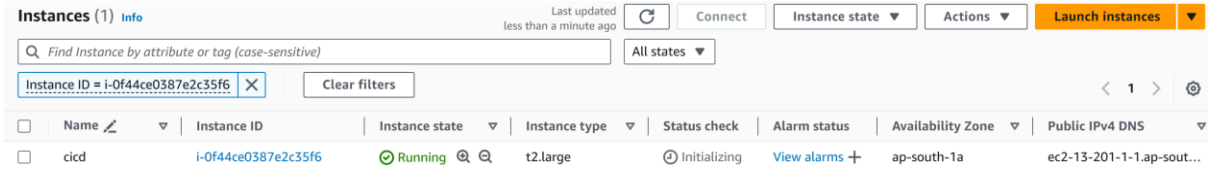
```
→  spring-boot-app git:(main) ✗ docker run -d -p 8010:8080 -t ultimate-cicd-pipeline:v1
8ec65093671787350fbcc2c025a4828f692b47698452511eec0e03fcc2fee28a
```

Access the application using port 8010.

CICD Lab:

Create an Ubuntu based AMI EC2 instance with t2 large - 2 cpus and 8 GB. Free tier one will not be able to handle as we are installing multiple tools. It will not cost much for one two hours.

Also create a key pair for the instance. RSA → .pem



```
yvaru@Tej MINGW64 ~/Downloads
$ chmod 400 "cicd.pem"
```

```
yvaru@Tej MINGW64 ~/Downloads
$ ls -al | grep cicd
-r--r--r-- 1 yvaru 197121      1678 Sep 25 11:23 cicd.pem
```

Now connect with EC2 instance using SSH and pem file.

```
yvaru@Tej MINGW64 ~/Downloads
$ ssh -i cicd.pem ubuntu@13.201.1.1
The authenticity of host '13.201.1.1 (13.201.1.1)' can't be established.
ED25519 key fingerprint is SHA256:5vOPWxLi/wJaAyuoGO9NkvLIUvBLwdS21IMJewqcBfk.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '13.201.1.1' (ED25519) to the list of known hosts.
Welcome to Ubuntu 24.04 LTS (GNU/Linux 6.8.0-1012-aws x86_64)
```

Now we are connected to EC2 Instance.

We have to install Java (JDK) which is pre-requisite to Jenkins.

Follow the Readme page here: https://github.com/VarunTej06/Jenkins-End-to-End/blob/main/README.md

```
ubuntu@ip-172-31-33-57:~$ sudo apt update
sudo apt install openjdk-17-jre
Hit:1 http://ap-south-1.ec2.archive.ubuntu.com/ubuntu noble InRelease
Get:2 http://ap-south-1.ec2.archive.ubuntu.com/ubuntu noble-updates InRelease [126 kB]
Get:3 http://ap-south-1.ec2.archive.ubuntu.com/ubuntu noble-backports InRelease [126 kB]
Get:4 http://security.ubuntu.com/ubuntu noble-security InRelease [126 kB]
Get:5 http://ap-south-1.ec2.archive.ubuntu.com/ubuntu noble/universe amd64 Packages [15.0 MB]
```

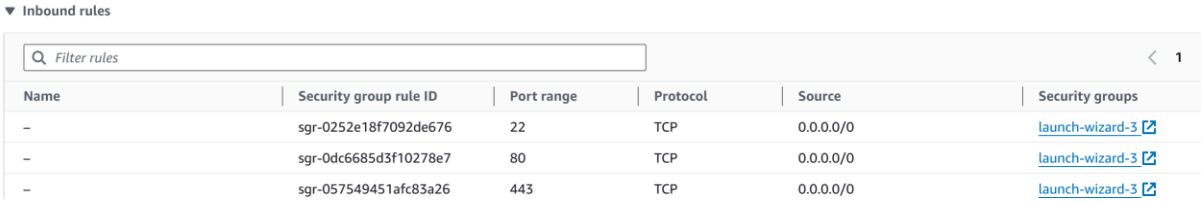
Check Java Version:

```
ubuntu@ip-172-31-33-57:~$ java -version
openjdk version "17.0.12" 2024-07-16
OpenJDK Runtime Environment (build 17.0.12+7-Ubuntu-1ubuntu224.04)
OpenJDK 64-Bit Server VM (build 17.0.12+7-Ubuntu-1ubuntu224.04, mixed mode, sharing)
```

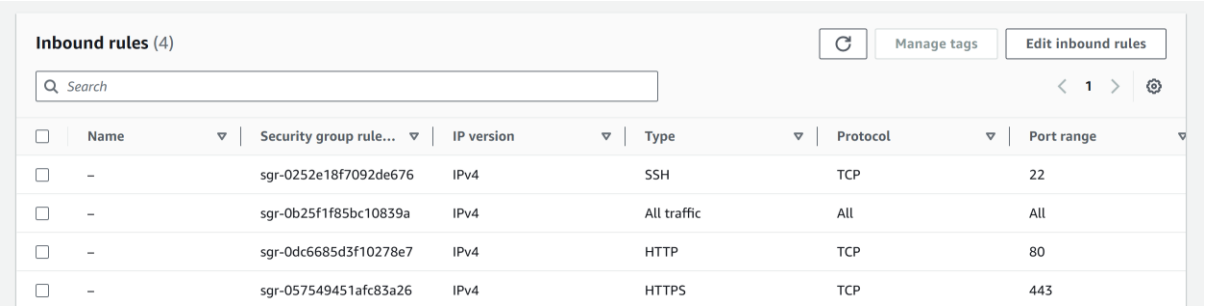
Now install jenkins:

```
ubuntu@ip-172-31-33-57:~$ curl -fsSL https://pkg.jenkins.io/debian/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 binary/ | sudo tee \
  /etc/apt/sources.list.d/jenkins.list > /dev/null
sudo apt-get update
sudo apt-get install jenkins
Hit:1 http://ap-south-1.ec2.archive.ubuntu.com/ubuntu noble InRelease
Ign:2 https://pkg.jenkins.io/debian binary/ InRelease
Hit:3 http://ap-south-1.ec2.archive.ubuntu.com/ubuntu noble-updates InRelease
Hit:4 http://ap-south-1.ec2.archive.ubuntu.com/ubuntu noble-backports InRelease
Get:5 https://pkg.jenkins.io/debian binary/ Release [2044 B]
```

Now allow inbound traffic to port 8080 to access Jenkins installed in Ec2 instance from our PC.



In our case I am adding 'All Traffic' since we will be installing any tools.



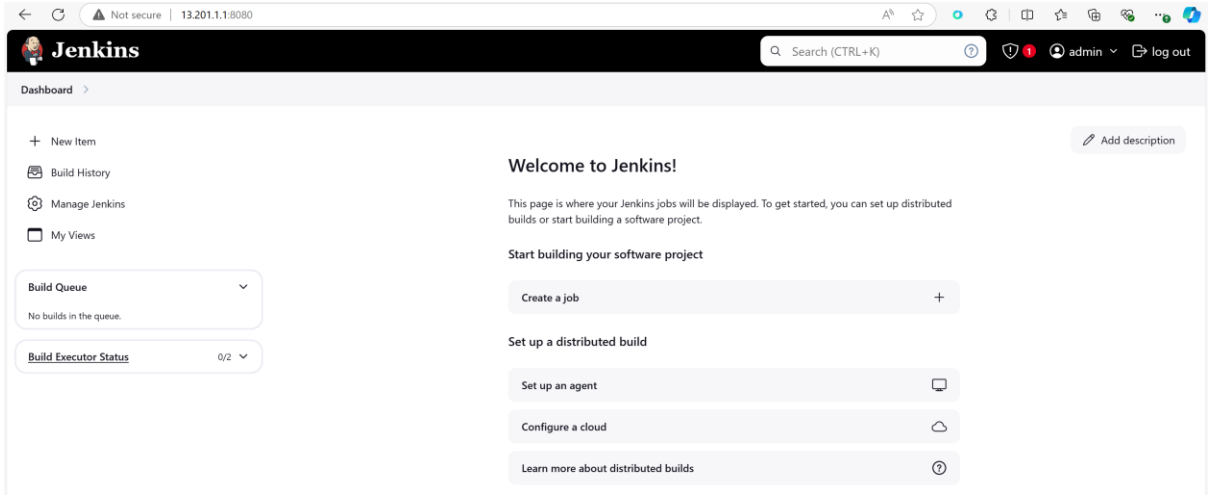
Jenkins is running now:

```
ubuntu@ip-172-31-33-57:~$ ps -ef | grep jenkins
jenkins     4652       1  1 06:18 ?        00:00:15 /usr/bin/java -Djava.awt.headless=true -jar /usr/share/java/jenkins.war --webroot=/var/cache/jenkins/war --
-httpPort=8080
ubuntu      4956    4925  0 06:41 pts/2    00:00:00 grep --color=auto jenkins
ubuntu@ip-172-31-33-57:~$
```

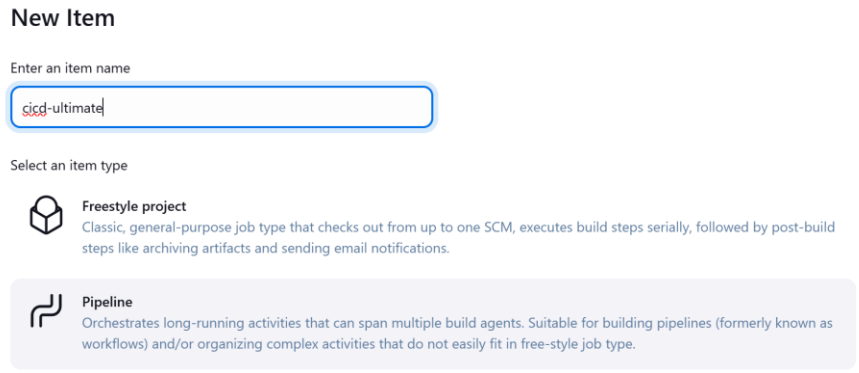
Now try to run the Jenkins in the browser: **public-IP:8080**

Paste the password and install suggested plugins → Skip and continue as Admin → Save and Finish.

Jenkins Dashboard:



Now we have to create pipeline in Jenkins. Create a pipeline project with name cicd-ultimate.



We can write the pipeline script in Groovy in Jenkins itself or we can take our jenkins pipeline script from GitHub also. We will prefer the second option.

Jenkinsfile in the repo should be inside the root of the application folder not in the root of the repo. As some repos will have multiple applications. It entirely depends on Project configuration mostly.

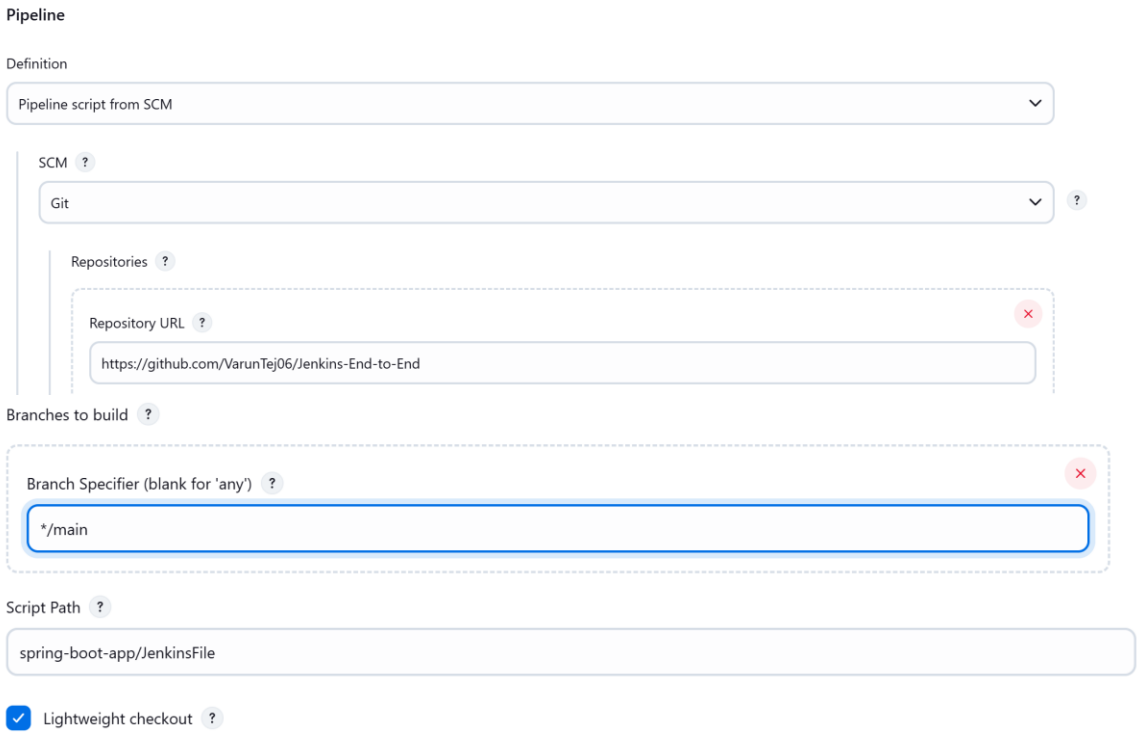
Our jenkins file here: https://github.com/VarunTej06/Jenkins-End-to-End/blob/main/spring-boot-app/JenkinsFile inside the application folder.

Now, we have to tell Jenkins where the Jenkins file is located.

In the Pipeline section give Git as SCM. It will be same for all SCM's. Then paste the URL to Repo.

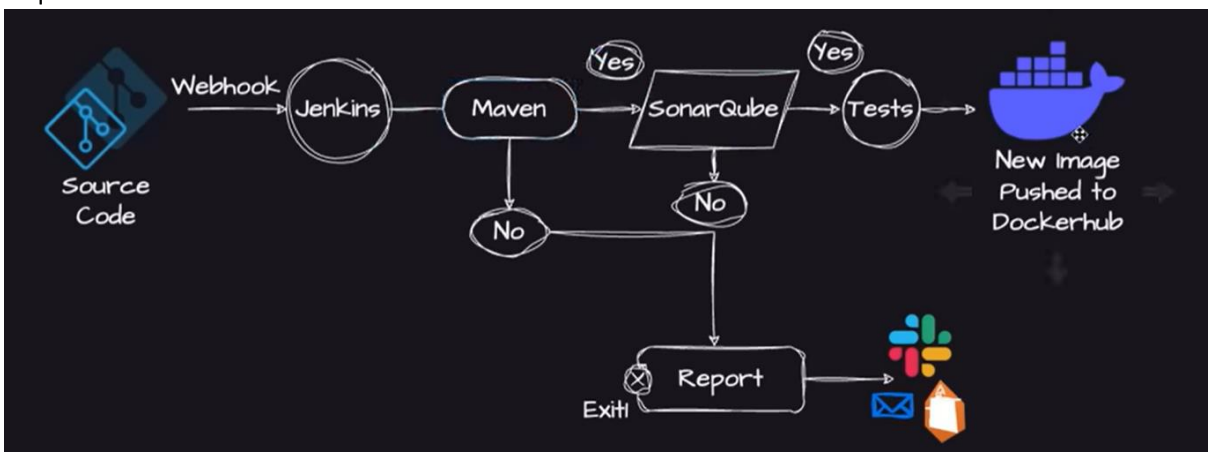
Then add the branch to build.

Then specify the path to the jenkisfile inside the repo in **Script Path.**



Then click on Save.

Now we have a Jenkins pipeline for which we are takign script from SCM and executes all the actions required in CI.



Before executing the pipeline we need to install all the tools first that are required.

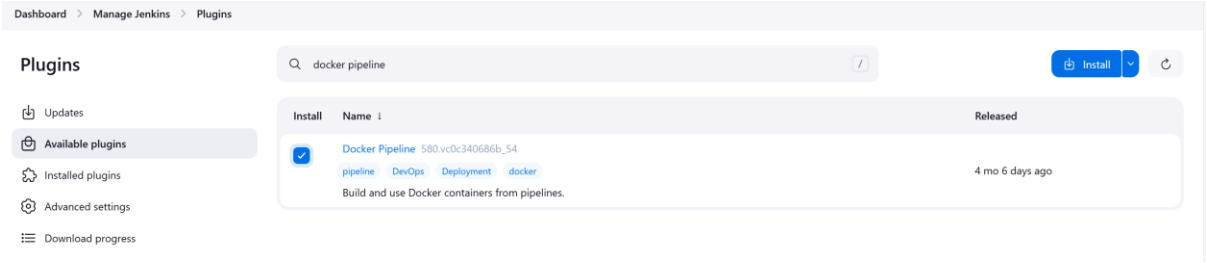
In the pipeline we are using docker as agent because if we are using another EC2 instance to do the actions mentioned in pipeline by configuring it as a slave to the Jenkins server then there will be some issues like we will not be using the Ec2 sinatnce all time if there is no dev code changes then unnecessarily we are spending money on that. Also it is not possible to Auto Scale.

```
pipeline {
  agent {
    docker {
      image 'abhishekf5/maven-abhishek-docker-agent:v1'
      args '--user root -v /var/run/docker.sock:/var/run/docker.sock' // mount Docker socket to access the host's Docker daemon
    }
  }
```

That's why we will be using Docker as agent then it will take care about creating a docker conatiner and execute all the steps we mentioned in th epipeline and the deletes the container at the last.

Since, we decided to use Docker as a agent we need to install docker plugin and choose the image wisely. Image we can take any existing as well. Abhishek had created an Image with Maven and Docker and it is in Docker hub available to everyone. We can use that.

For Plugin: Go to Dashboard → Manage jenkins → Plugins → Search for Docker Pipeline



Click on Install. Docker plugin is installed. We no need to install Maven as we are using Docker image which already had Maven in it. Check the Jenkins file:

```
pipeline {
  agent {
    docker {
      image 'abhishekf5/maven-abhishek-docker-agent:v1'
      args '--user root -v /var/run/docker.sock:/var/run/docker.sock' // mount Docker socket to access the host's Docker daemon
    }
  }
```

We have to install Sonar server and sonar plugin. Lets install the sonar plugin first.

Now, we have to install the Sonar server on the EC2 instance:

Use this Readme file to configure Sonarqubes in EC2: https://github.com/VarunTej06/Jenkins-End-to-End/blob/main/README.md

Configure a Sonar Server locally

```
apt install unzip
adduser sonarqube
wget https://binaries.sonarsource.com/Distribution/sonarqube/sonarqube-9.4.0.54424.zip
unzip *
chmod -R 755 /home/sonarqube/sonarqube-9.4.0.54424
chown -R sonarqube:sonarqube /home/sonarqube/sonarqube-9.4.0.54424
cd sonarqube-9.4.0.54424/bin/linux-x86-64/
./sonar.sh start
```

Login to root user in the Ec2 instance as we need to create new user and install package called unzip.

```
ubuntu@ip-172-31-33-57:~$ sudo -i
root@ip-172-31-33-57:~# pwd
/root
root@ip-172-31-33-57:~# apt install unzip
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
```

Add user and password:

```
root@ip-172-31-33-57:~# adduser sonarqube
info: Adding user `sonarqube' ...
info: Selecting UID/GID from range 1000 to 59999 ...
info: Adding new group `sonarqube' (1001) ...
info: Adding new user `sonarqube' (1001) with group `sonarqube (1001)' ...
info: Creating home directory `/home/sonarqube' ...
info: Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for sonarqube
Enter the new value, or press ENTER for the default
        Full Name []:
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n] Y
info: Adding new user `sonarqube' to supplemental / extra groups `users' ...
info: Adding user `sonarqube' to group `users' ...
root@ip-172-31-33-57:~#
```

Login to the sonarqube user and download zip file.

```
root@ip-172-31-33-57:~# su - sonarqube
sonarqube@ip-172-31-33-57:~$ wget https://binaries.sonarsource.com/Distribution/sonarqube/sonarqube-9.4.0.54424.zip
--2024-09-25 07:47:47--  https://binaries.sonarsource.com/Distribution/sonarqube/sonarqube-9.4.0.54424.zip
Resolving binaries.sonarsource.com (binaries.sonarsource.com)... 54.182.0.125, 54.182.0.53, 54.182.0.74, ...
Connecting to binaries.sonarsource.com (binaries.sonarsource.com)|54.182.0.125|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 287666040 (274M) [binary/octet-stream]
Saving to: 'sonarqube-9.4.0.54424.zip'

sonarqube-9.4.0.54424.zip           100%[===================================================================>] 274.34M   116MB/s    in 2.4s

2024-09-25 07:47:50 (116 MB/s) - 'sonarqube-9.4.0.54424.zip' saved [287666040/287666040]
```

```
sonarqube@ip-172-31-33-57:~$ unzip *
Archive:  sonarqube-9.4.0.54424.zip
   creating: sonarqube-9.4.0.54424/
  inflating: sonarqube-9.4.0.54424/dependency-license.json
   creating: sonarqube-9.4.0.54424/bin/
   creating: sonarqube-9.4.0.54424/bin/jsw-license/
```

Sonarqube folder is extracted.

```
sonarqube@ip-172-31-33-57:~$ ls
sonarqube-9.4.0.54424  sonarqube-9.4.0.54424.zip
sonarqube@ip-172-31-33-57:~$
```

Then grant the permissions:

```
sonarqube@ip-172-31-33-57:~$ chmod -R 755 /home/sonarqube/sonarqube-9.4.0.54424
sonarqube@ip-172-31-33-57:~$ chown -R sonarqube:sonarqube /home/sonarqube/sonarqube-9.4.0.54424
sonarqube@ip-172-31-33-57:~$ cd sonarqube-9.4.0.54424/bin/linux-x86-64/
sonarqube@ip-172-31-33-57:~/sonarqube-9.4.0.54424/bin/linux-x86-64$
```

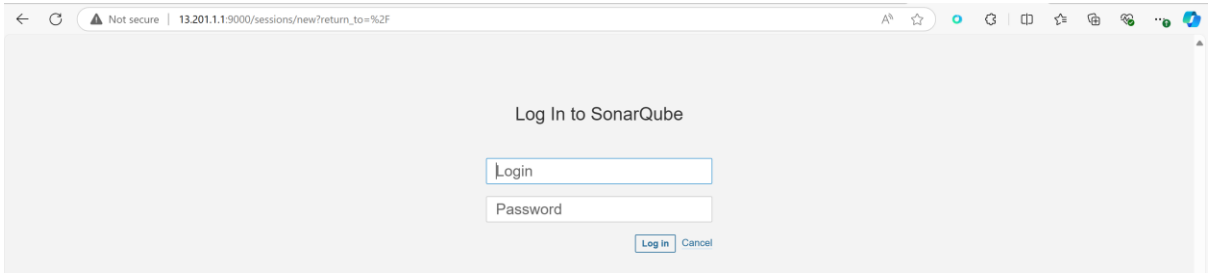
Then go to the folder depending upon the architecture. In our case we are using EC2 instance with Linux so we will choose linux-x86-64.

```
sonarqube@ip-172-31-33-57:~/sonarqube-9.4.0.54424/bin/linux-x86-64$ cd ..
sonarqube@ip-172-31-33-57:~/sonarqube-9.4.0.54424/bin$ ls
jsw-license  linux-x86-64  macosx-universal-64  windows-x86-64
sonarqube@ip-172-31-33-57:~/sonarqube-9.4.0.54424/bin$ cd linux-x86-64/
sonarqube@ip-172-31-33-57:~/sonarqube-9.4.0.54424/bin/linux-x86-64$
```

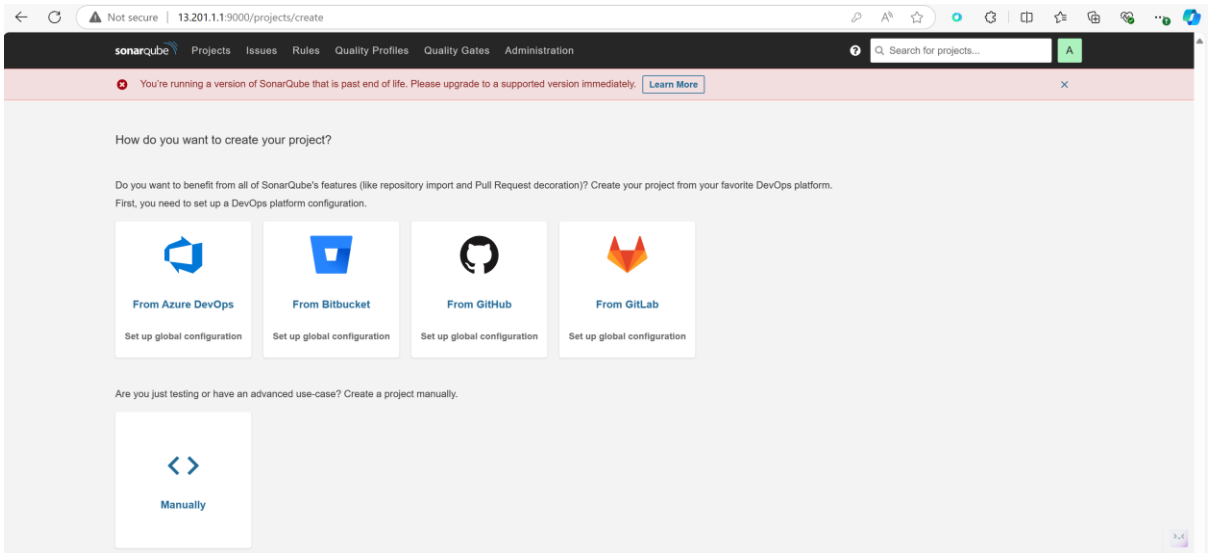
Now start the Sonar server:

```
sonarqube@ip-172-31-33-57:~/sonarqube-9.4.0.54424/bin/linux-x86-64$ ls
lib  sonar.sh  wrapper
sonarqube@ip-172-31-33-57:~/sonarqube-9.4.0.54424/bin/linux-x86-64$ ./sonar.sh start
Starting SonarQube...
Started SonarQube.
sonarqube@ip-172-31-33-57:~/sonarqube-9.4.0.54424/bin/linux-x86-64$
```

We no need to edit any Inbound traffic to the EC2 instace as we already gave All traffic. By default, Sonar server will start on port 9000.

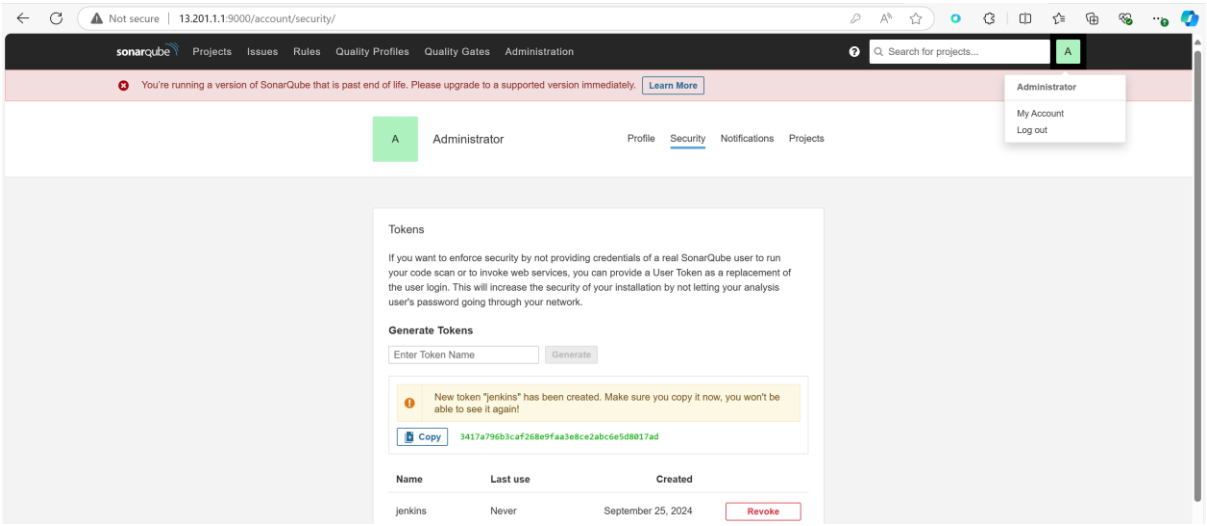


admin is password and username. Asks for reset password. Reset It. Kept as user.

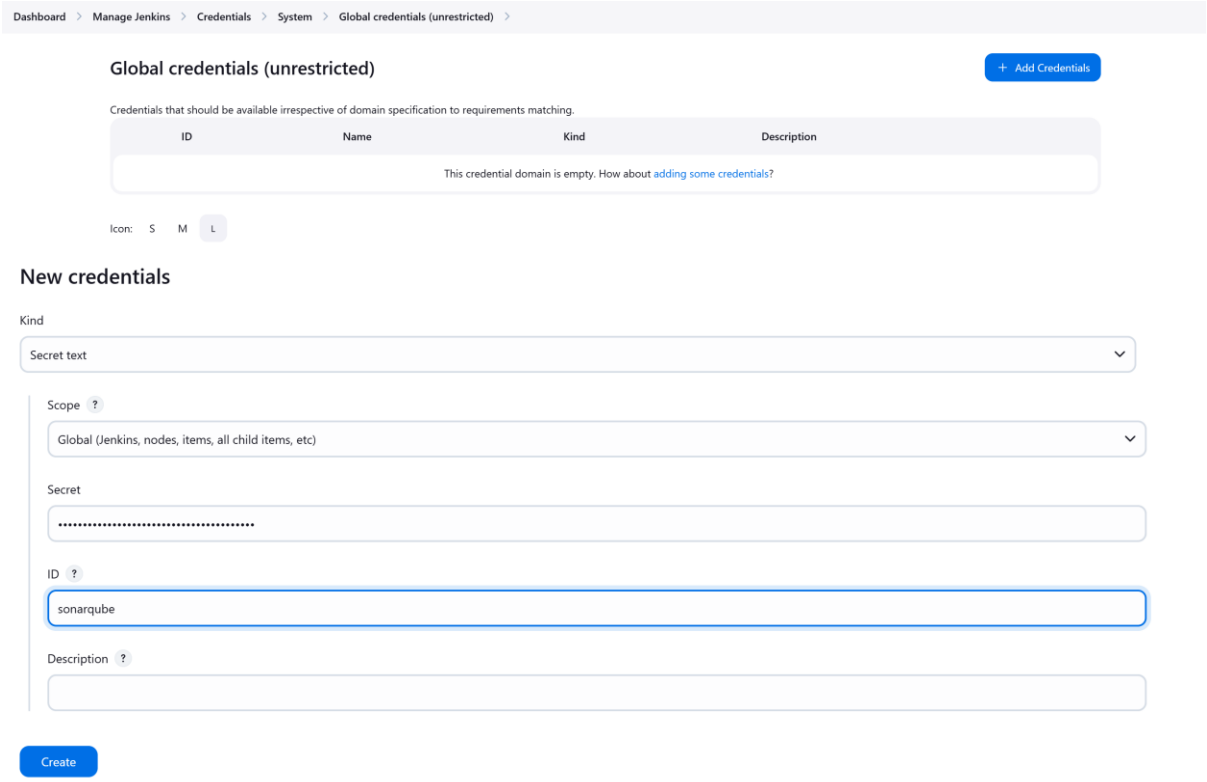


Now, Jenkins and Sonarqubes are two different applications, then how will jenkins pipeline will authenticate the sonarqube.

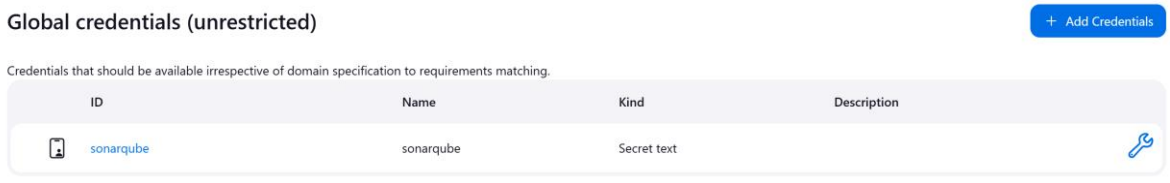
Generate Token form Sonarqubes:

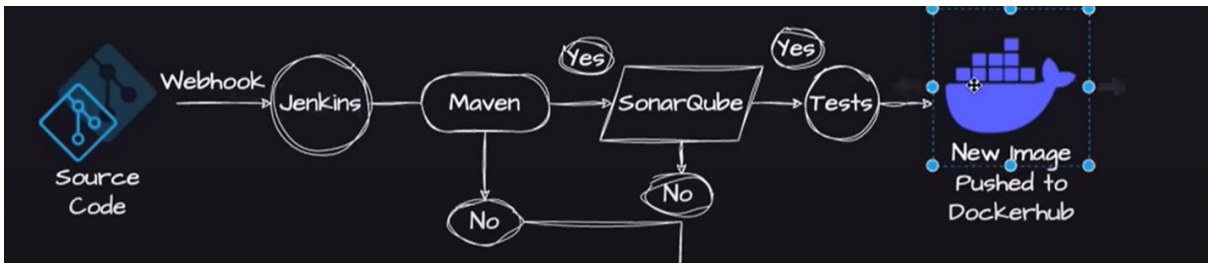


Now, go to Jenkins → Manage Jenkins → Security → Credentials → System → Global Credentails → Add Credentails → Add the Token here.



Later we will use this in our pipeline.





Now, on EC2 instnace we have to install docker.

Go back to the root user in EC2 instance:

```
sonarqube@ip-172-31-33-57:~$ exit
logout
ubuntu@ip-172-31-33-57:~$ sudo -i
root@ip-172-31-33-57:~#
```

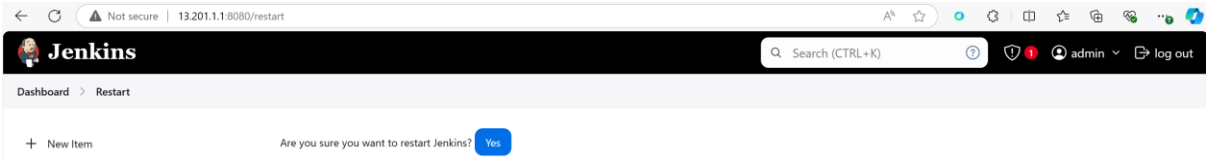
Now, install Docker: Take help form same Readme file for commands:

```
root@ip-172-31-33-57:~# sudo apt install docker.io
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  bridge-utils containerd dns-root-data dnsmasq-base pigz runc ubuntu-fan
Suggested packages:
  ifupdown aufs-tools cgroupfs-mount | cgroup-lite debootstrap docker-buildx docker-compose-v2 docker-doc rinse zfs-fuse | zfsutils
```

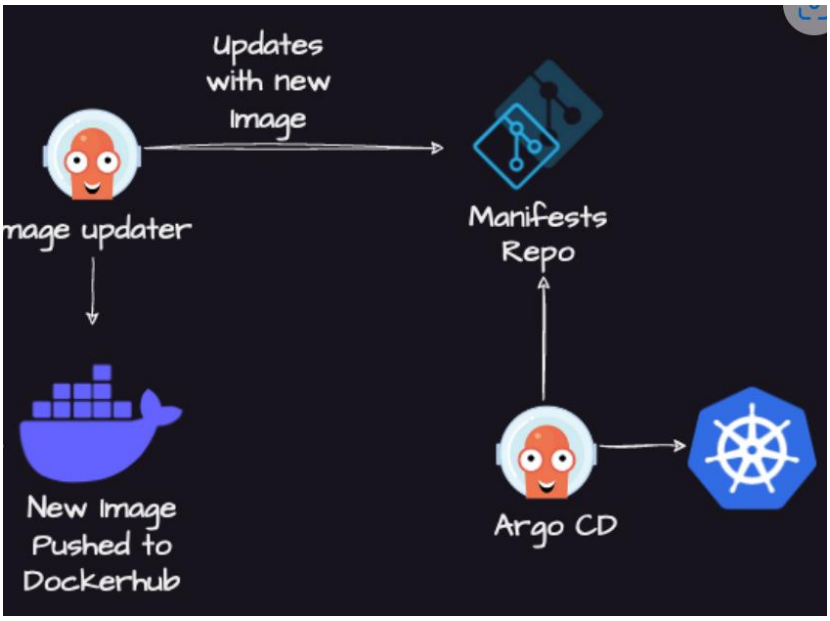
Grant Jenkins user and Ubuntu user permission to docker deamon.

```
root@ip-172-31-33-57:~# usermod -aG docker jenkins
usermod -aG docker ubuntu
systemctl restart docker
root@ip-172-31-33-57:~#
```

Now, restart the Jenkins once.



Now, CI installation parts are done. Lets do for CD part now.



In you local PC we need to setup minikube cluster as EC2 instance is installed with multiple tools.

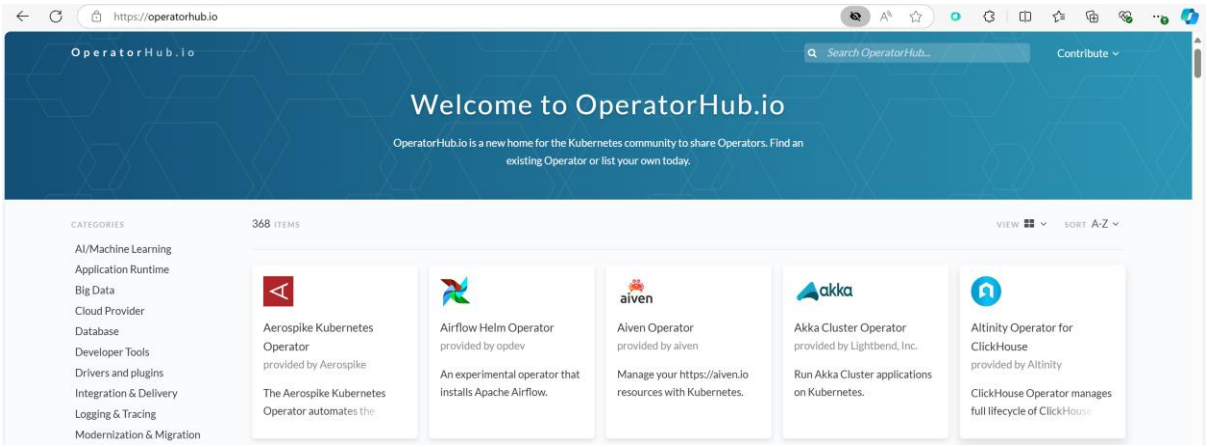
Install minikube, kubectl and docker which works as a driver to minikube for creating cluster in your local and create a cluster.

```
varun@DevOps:~$ minikube start --memory=4098 --driver=docker
😄  minikube v1.34.0 on Ubuntu 24.04
✨  Using the docker driver based on user configuration
📌  Using Docker driver with root privileges
👍  Starting "minikube" primary control-plane node in "minikube" cluster
🚜  Pulling base image v0.0.45 ...
🔥  Creating docker container (CPUs=2, Memory=4098MB) ...
🐳  Preparing Kubernetes v1.31.0 on Docker 27.2.0 ...
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔗  Configuring bridge CNI (Container Networking Interface) ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
varun@DevOps:~$
```

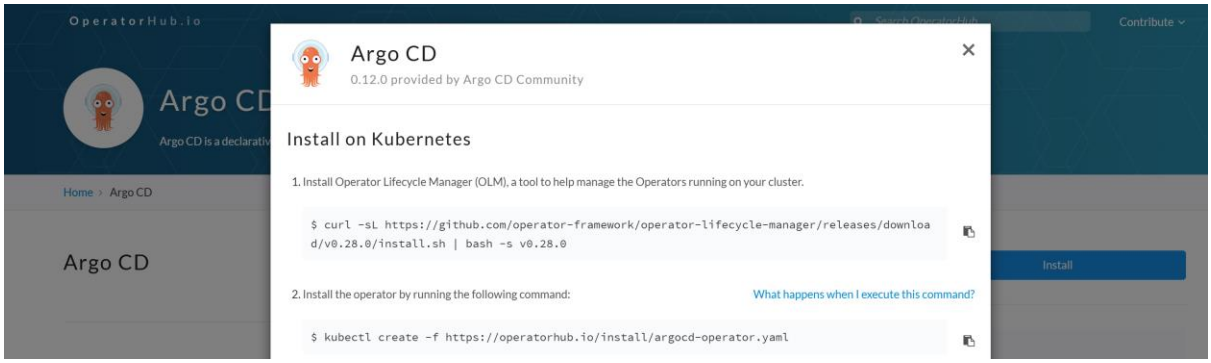
Cluster is created now. Now we have to install Argo CD in the cluster.

In K8s cluster when ever we want to install any Controller, we need to make use of K8s Operators which will manage the lifecycle of K8s controllers.

We can install K8s controller using K8s operators. Take help from here: https://operatorhub.io/



Search for ArgoCd and click on Install.



Execute all commands.

```
varun@DevOps:~$ curl -sL https://github.com/operator-framework/operator-lifecycle-manager/releases/download/v0.28.0/install.sh | bash -s v0.28.0
customresourcedefinition.apiextensions.k8s.io/catalogsources.operators.coreos.com created
customresourcedefinition.apiextensions.k8s.io/clusterserviceversions.operators.coreos.com created
customresourcedefinition.apiextensions.k8s.io/installplans.operators.coreos.com created
customresourcedefinition.apiextensions.k8s.io/olmconfigs.operators.coreos.com created
customresourcedefinition.apiextensions.k8s.io/operatorconditions.operators.coreos.com created
customresourcedefinition.apiextensions.k8s.io/operatorgroups.operators.coreos.com created
customresourcedefinition.apiextensions.k8s.io/operators.operators.coreos.com created
varun@DevOps:~$ kubectl create -f https://operatorhub.io/install/argocd-operator.yaml
subscription.operators.coreos.com/my-argocd-operator created
varun@DevOps:~$
varun@DevOps:~$ kubectl get csv -n operators
NAME                  DISPLAY   VERSION   REPLACES                  PHASE
argocd-operator.v0.12.0   Argo CD   0.12.0    argocd-operator.v0.11.0   Succeeded
varun@DevOps:~$ kubectl get pods -n operators
NAME                                        READY   STATUS    RESTARTS   AGE
argocd-operator-controller-manager-745f7fb9cf-c69xd   1/1     Running   0          3m51s
```

Argo CD operator is running now.

Understanding the Jenkins File:

First stage is checkout but in our cas eit is not needed. Because, jenkins is already doing the checkout as the Jenkinsfile is also stored in the same repository.

If we want to write the Jenkins file in Jenkins then we need to add the checkout stage.

```
stages {
  stage('Checkout') {
    steps {
      sh 'echo passed'
      //git branch: 'main', url: 'https://github.com/VarunTej06/Jenkins-End-to-End.git'
    }
  }
```

So, it is commented.

Next stage is Build and Test: No need to add steps for installing maven as we are taking maven in image already in Agent.

```
stage('Build and Test') {
  steps {
    sh 'ls -ltr'
    // build the project and create a JAR file
    sh 'cd spring-boot-app && mvn clean package'
  }
}
```

What is the difference between mvn clean package and mvn clean install?

If we want to push the entrprise archieve/ JAR/ WAR archieve to the artifactory then we will use mvn clean install. We do not want to publish the artifactory any where instead we want to use it in the Docker image and then we will publish the docker image into image artifactory.

This mvn clean package will check for POM.XML file. This is already there in repo. This is repsonsible for building the application by getting all dependencies. Dependencies are huge we cannot push them to repo. So, we will use POM.XML file. Wherever we are using this application it will use this XML file to build the application.

So, that's why we are changing the directory location and trying to build the application in this stage.

After this stage artifact will be generated. We can see that file inside the target folder.

```
→  spring-boot-app git:(main) ✗ ls
Dockerfile README.md  pom.xml    src       target
→  spring-boot-app git:(main) ✗ ls target
classes                 maven-archiver          spring-boot-web.jar
generated-sources       maven-status            spring-boot-web.jar.original
→  spring-boot-app git:(main) ✗
```

This picture is from local. We run the application once in local.

Dockerfile will use this artifact later and start the application in the container.

Next stage is Static Code Analysis:

```
stage('Static Code Analysis') {
  environment {
    SONAR_URL = "http://13.201.1.1:9000"
  }
  steps {
    withCredentials([string(credentialsId: 'sonarqube', variable: 'SONAR_AUTH_TOKEN')]) {
      sh 'cd spring-boot-app && mvn sonar:sonar -Dsonar.login=$SONAR_AUTH_TOKEN -Dsonar.host.url=${SONAR_URL}'
    }
  }
}
```

We need to hardcode the sonarqube URL here. We used another maven target here called mvn sonar to execute sonar for static code analysis. For that it needs URL and Authentication Token.

Next stage is steps to build and push docker image:

```
stage('Build and Push Docker Image') {
  environment {
    DOCKER_IMAGE = "varuntej0620/ultimate-cicd:${BUILD_NUMBER}"
    // DOCKERFILE_LOCATION = "spring-boot-app/Dockerfile"
    REGISTRY_CREDENTIALS = credentials('docker-cred')
  }
  steps {
    script {
        sh 'cd spring-boot-app && docker build -t ${DOCKER_IMAGE} .'
        def dockerImage = docker.image("${DOCKER_IMAGE}")
        docker.withRegistry('https://index.docker.io/v1/', "docker-cred") {
            dockerImage.push()
        }
    }
  }
}
```

Here also we need to pass dockerhub credentails.

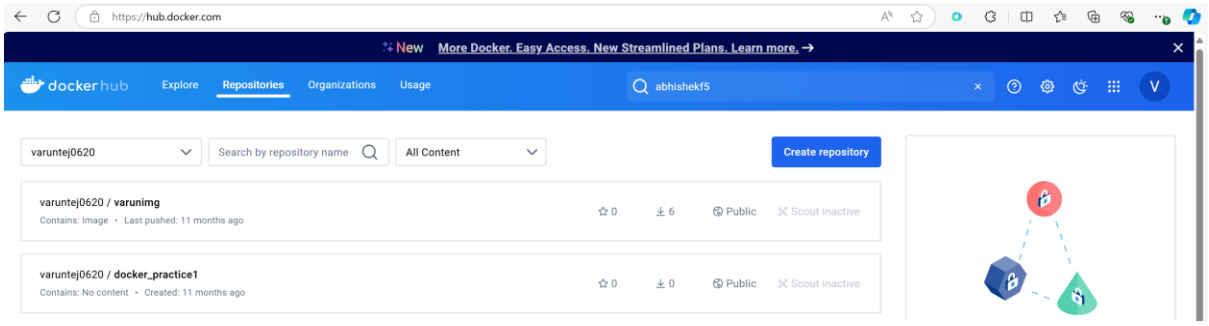
stage('Build and Push Docker Image'): This defines a pipeline stage named "Build and Push Docker Image."

1. **environment { ... }**:
    - DOCKER_IMAGE: Specifies the Docker image name along with a tag, using the Jenkins build number (${BUILD_NUMBER}) for versioning.
    - REGISTRY_CREDENTIALS: Fetches Docker credentials stored in Jenkins with the ID docker-cred.
2. **steps { ... }**:
   Inside the steps block, the following happens:
    - **sh 'cd spring-boot-app && docker build -t ${DOCKER_IMAGE} .'**: This shell command navigates to the spring-boot-app directory and builds a Docker image using the Dockerfile in that directory. It tags the image as specified in DOCKER_IMAGE.
    - **def dockerImage = docker.image("${DOCKER_IMAGE}")**: A reference to the newly built Docker image is stored in the dockerImage variable.
    - **docker.withRegistry(...) { ... }**: This block logs in to Docker Hub using the credentials (docker-cred). Once authenticated, it pushes the built image to Docker Hub using dockerImage.push().

Improvements or Notes:

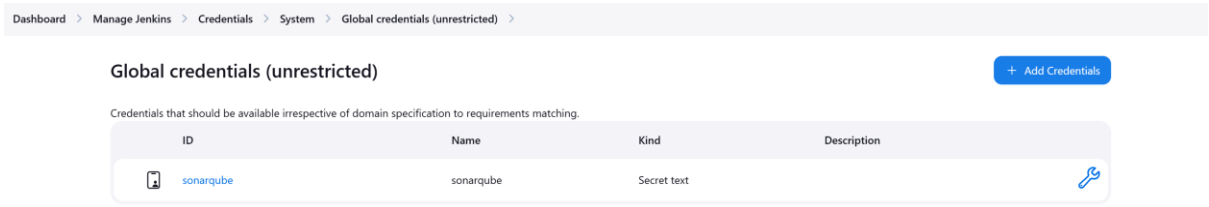
- The commented-out line // DOCKERFILE_LOCATION = "spring-boot-app/Dockerfile" suggests that you might want to explicitly define the location of the Dockerfile, but it seems the default . path is being used, which works fine if the Dockerfile is in the root of spring-boot-app.
- Ensure that the credentials (docker-cred) are correctly set up in Jenkins to avoid authentication issues during the push.

DockerHub: https://hub.docker.com/

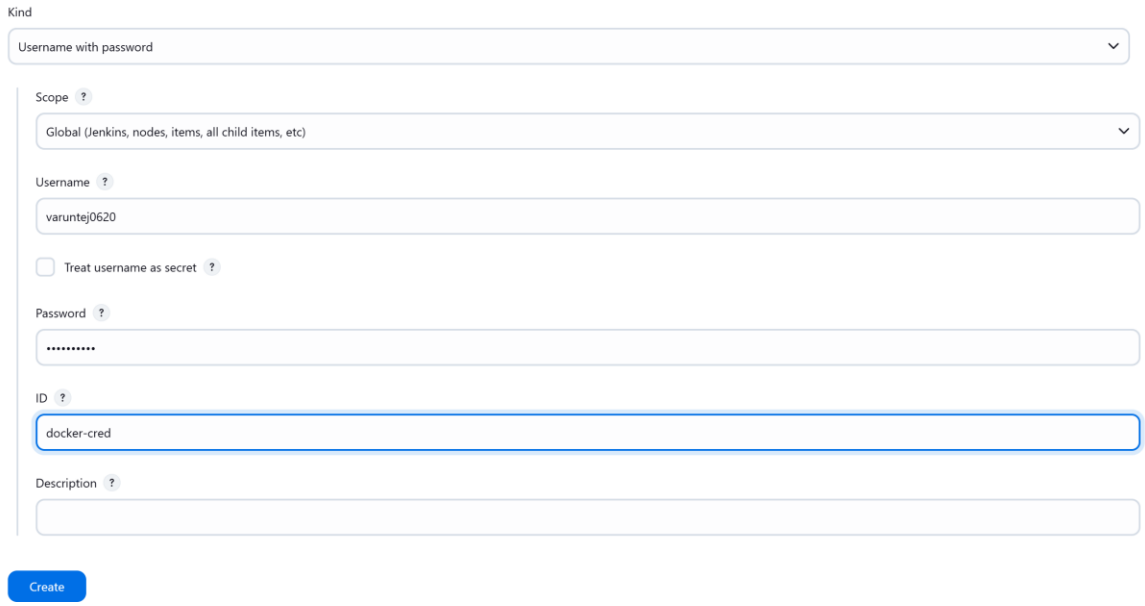


Whatever Image you push it will be added here.

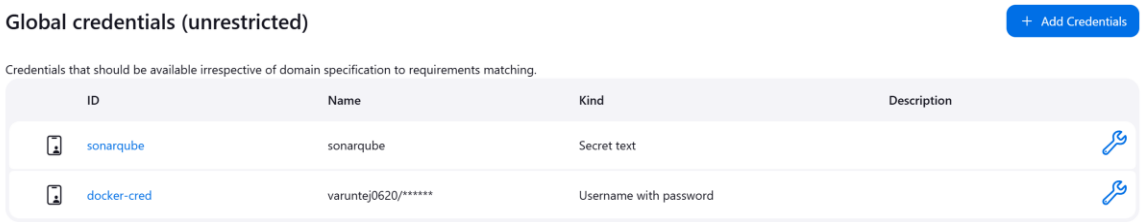
Adding Docker credentails in jenkins:



Click on Add credentials and add Username, password:



ID should be same as the ID mentioned in Jenkinsfile in SCM.



Next stage is updating the deployment file with latest version. We can use Argo Image updater or Shell Script. In our case we used shell script.

For these manigets files also we need to store in separate repo but we are using same git repo but storing the files in separate directory.

```
    stage('Update Deployment File') {
        environment {
            GIT_REPO_NAME = "Jenkins-End-to-End"
            GIT_USER_NAME = "VarunTej06"
        }
        steps {
            withCredentials([string(credentialsId: 'github', variable: 'GITHUB_TOKEN')]) {
                sh '''
                    git config user.email "varuntej1045@gmail.com"
                    git config user.name "Varun Teja"
                    BUILD_NUMBER=${BUILD_NUMBER}
                    sed -i "s/replaceImageTag/${BUILD_NUMBER}/g" spring-boot-app-manifests/deployment.yml
                    git add spring-boot-app-manifests/deployment.yml
                    git commit -m "Update deployment image to version ${BUILD_NUMBER}"
                    git push https://${GITHUB_TOKEN}@github.com/${GIT_USER_NAME}/${GIT_REPO_NAME} HEAD:main
                '''
            }
        }
    }
  }
}
```

Stage Breakdown:

1. **environment block**:
    - GIT_REPO_NAME: Specifies the GitHub repository name.
    - GIT_USER_NAME: Specifies the GitHub username.
2. **withCredentials([string(credentialsId: 'github', variable: 'GITHUB_TOKEN')]) { ... }**:
    - This block uses the withCredentials step to retrieve a GitHub Personal Access Token (PAT) stored in Jenkins under the credentials ID github.

- The token is temporarily stored in the GITHUB_TOKEN environment variable for secure use during the Git operations.

3. **Shell commands in sh ''' ... ''' block**:
    - **git config user.email "varuntej1045@gmail.com"**: Configures the email for Git commits.
    - **git config user.name "Varun Teja"**: Configures the username for Git commits.
    - **sed -i "s/replaceImageTag/${BUILD_NUMBER}/g" spring-boot-app-manifests/deployment.yml**:
        - This uses the sed command to replace the placeholder replaceImageTag in the deployment.yml file with the actual BUILD_NUMBER (i.e., the current Jenkins build number). This ensures that the deployment file gets updated with the correct Docker image tag.
    - **git add spring-boot-app-manifests/deployment.yml**: Stages the modified deployment file.
    - **git commit -m "Update deployment image to version ${BUILD_NUMBER}"**: Commits the changes with a message indicating the build number.
    - **git push https://${GITHUB_TOKEN}@github.com/${GIT_USER_NAME}/${GIT_REPO_NAME} HEAD:main**:
        - This pushes the commit to the main branch of the GitHub repository.
        - The GitHub token (${GITHUB_TOKEN}) is securely passed in the URL for authentication without exposing the token.

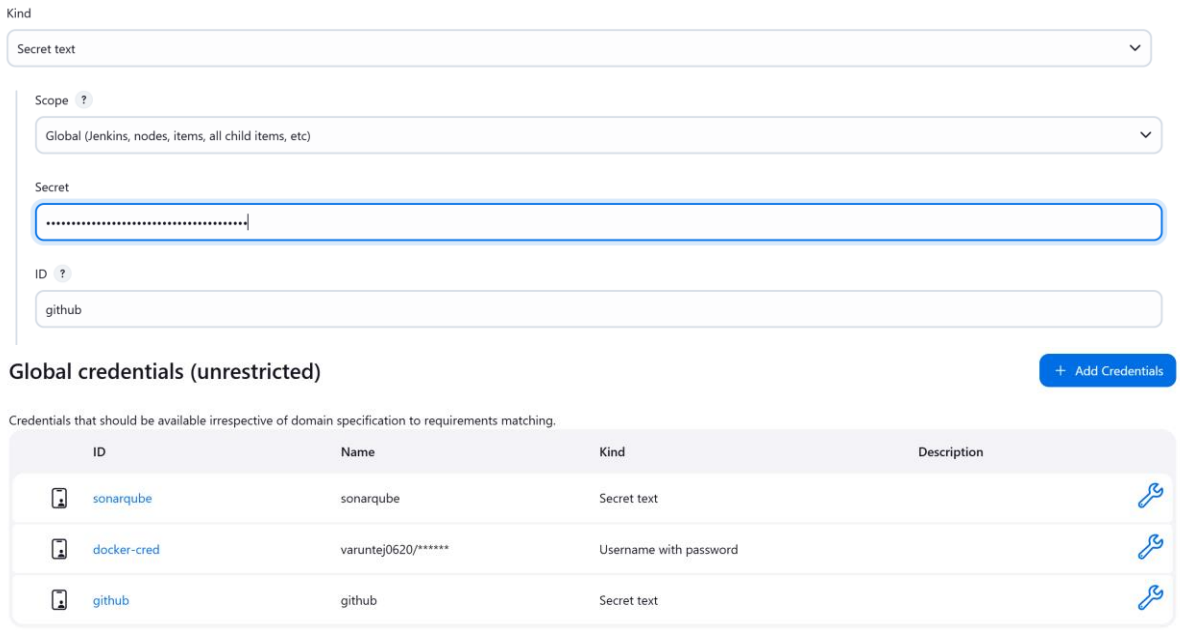
Key Points:

- **Credentials (GITHUB_TOKEN)**: Make sure you have a GitHub Personal Access Token stored in Jenkins under the ID github. You can generate a PAT from GitHub and set it up in Jenkins by following these steps:
    1. Go to your GitHub account, and navigate to **Settings** > **Developer settings** > **Personal access tokens**.
    2. Generate a new token with the necessary scopes (e.g., repo scope for pushing changes).
    3. Add this token in Jenkins under Manage Jenkins > Manage Credentials > Add Credentials as a Secret text, with ID set to github.

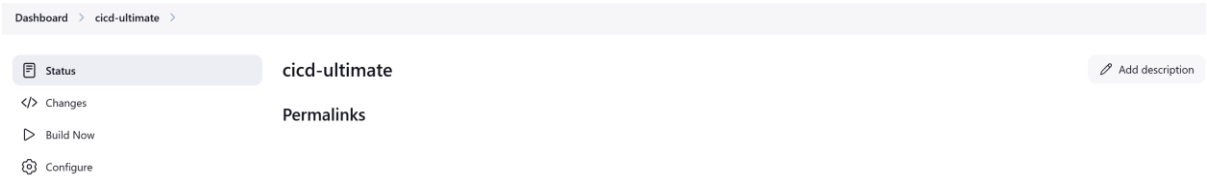
Adding GitHub credentials to jenkins:

GitHub moved from Username and passwords to Access token. We can use Secret text and add token here. Later Jenkins pipeline will use.

ID for this token to be mentioned same as ID mentioned in pipeline.



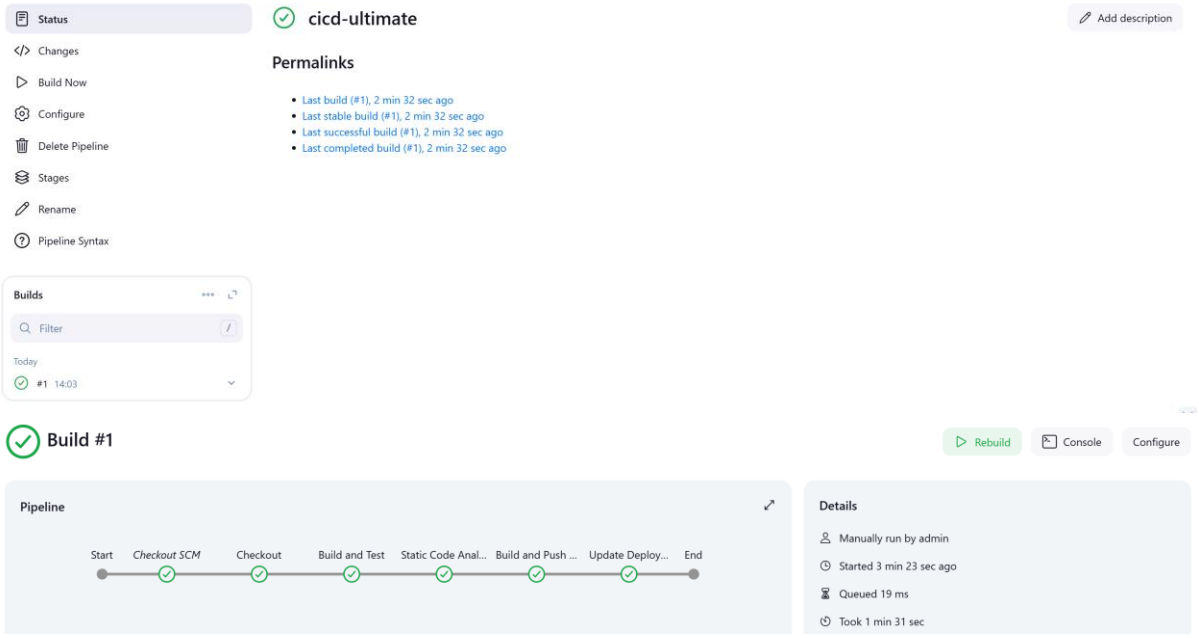
Restart Jenkins. And click on Build Now:



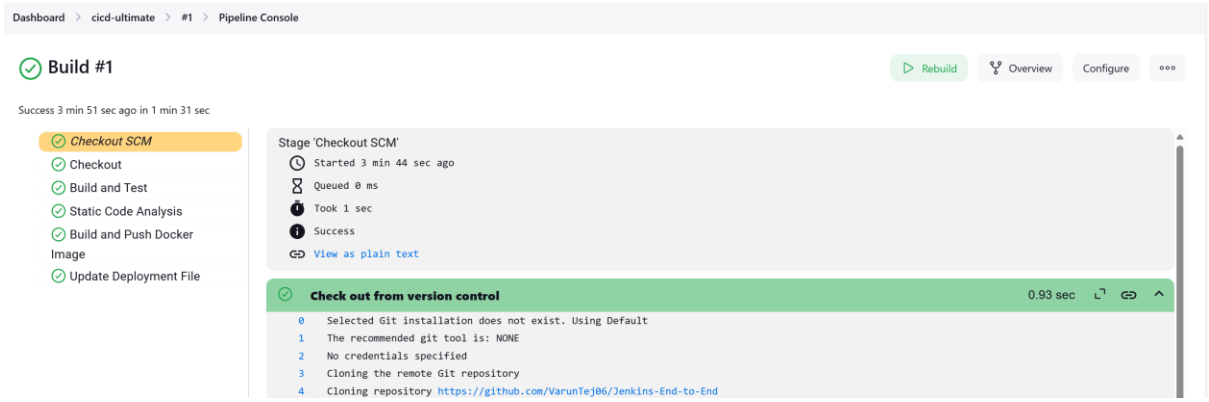
Lets check what is the image version before getting updated. (Here Username is not updated yet).

```
spec:
  containers:
  - name: spring-boot-app
    image: abhishekf5/ultimate-cicd:replaceImageTag
    ports:
    - containerPort: 8080
```

After the build is success:



Check pipeline console for detailed logs of each stage:

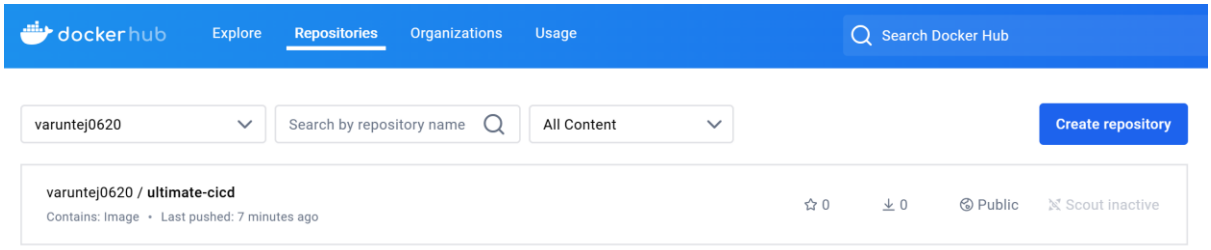


Lets check what is the image version in deployment.yaml after build is success.

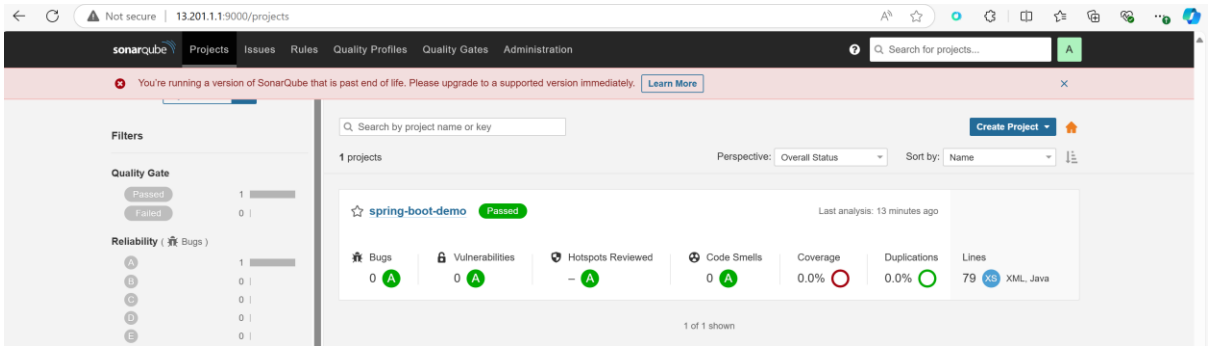
```
spec:
  containers:
  - name: spring-boot-app
    image: varuntej0620/ultimate-cicd:1
    ports:
    - containerPort: 8080
```

Note: If you want to retry the project replace the version number with replaceImageTag again.

Image uploaded to DockerHub as well.



Let's check report is pushed to Sonar or not.

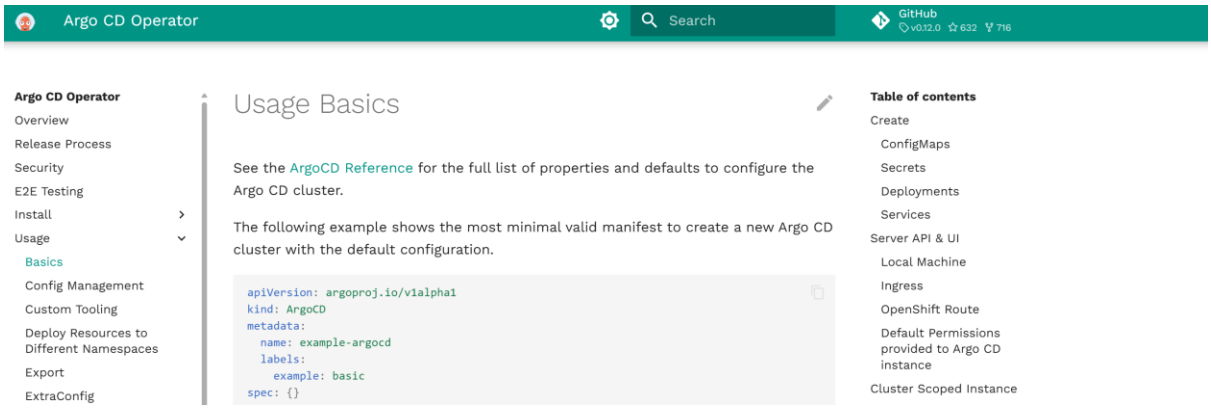


If we add unit test Coverage will increase.

CI part is done. We left with CD part using Argo CD.

Go to Operator Documentation: https://argocd-operator.readthedocs.io/en/latest/usage/basics/

Copy this part:



And paste it in the argocd-basic.yml file. This is for creating argocd controller. This file cretaes it.

```
varun@DevOps:~$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured

varun@DevOps:~$ vim argocd-basic.yml
varun@DevOps:~$ kubectl apply -f argocd-basic.yml
Warning: ArgoCD v1alpha1 version is deprecated and will be converted to v1beta1 automatically. Moving forward, please use v1beta1 as the ArgoCD A
argocd.argoproj.io/example-argocd created
```

```
varun@DevOps:~$ kubectl get pods
NAME                                    READY   STATUS    RESTARTS   AGE
example-argocd-application-controller-0   1/1     Running   0          5m37s
example-argocd-redis-6545fd6d6c-vtzs4     1/1     Running   0          5m37s
example-argocd-repo-server-869d5757c7-cqsr8   1/1     Running   0          5m37s
example-argocd-server-76bb84cddc-v7ztl    1/1     Running   0          5m37s
```

```
varun@DevOps:~$ kubectl get svc
NAME                          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)             AGE
example-argocd-metrics        ClusterIP   10.102.61.166    <none>        8082/TCP            7m36s
example-argocd-redis          ClusterIP   10.104.167.158   <none>        6379/TCP            7m36s
example-argocd-repo-server    ClusterIP   10.98.123.186    <none>        8081/TCP,8084/TCP   7m36s
example-argocd-server         ClusterIP   10.100.237.41    <none>        80/TCP,443/TCP      7m36s
example-argocd-server-metrics ClusterIP   10.102.74.223    <none>        8083/TCP            7m36s
kubernetes                    ClusterIP   10.96.0.1        <none>        443/TCP             3h16m
```

This argocd-server is responsible for argocd UI. Edit the type to NodePort from ClusterIP.

```
varun@DevOps:~$ kubectl edit svc example-argocd-server
service/example-argocd-server edited
```

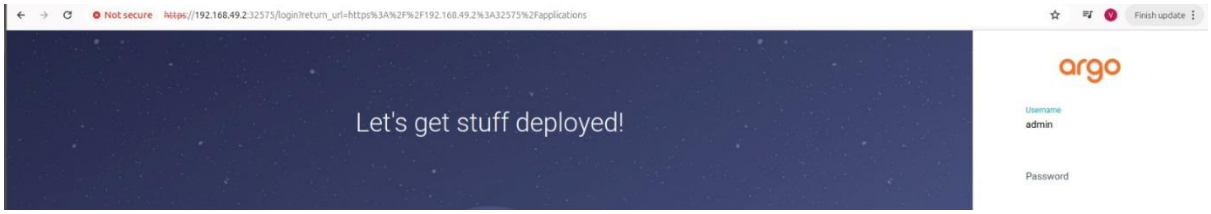
```
    protocol: TCP
    targetPort: 8080
  selector:
    app.kubernetes.io/name: example-argocd-server
  sessionAffinity: None
  type: NodePort
status:
  loadBalancer: {}
```

```
varun@DevOps:~$ kubectl get svc
NAME                          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
example-argocd-metrics        ClusterIP   10.102.61.166    <none>        8082/TCP                     66m
example-argocd-redis          ClusterIP   10.104.167.158   <none>        6379/TCP                     66m
example-argocd-repo-server    ClusterIP   10.98.123.186    <none>        8081/TCP,8084/TCP            66m
example-argocd-server         NodePort    10.100.237.41    <none>        80:32575/TCP,443:31406/TCP   66m
example-argocd-server-metrics ClusterIP   10.102.74.223    <none>        8083/TCP                     66m
kubernetes                    ClusterIP   10.96.0.1        <none>        443/TCP                      4h14m
```

You can get the URL of Argo CD server from here: agrocd pods should be up and running.

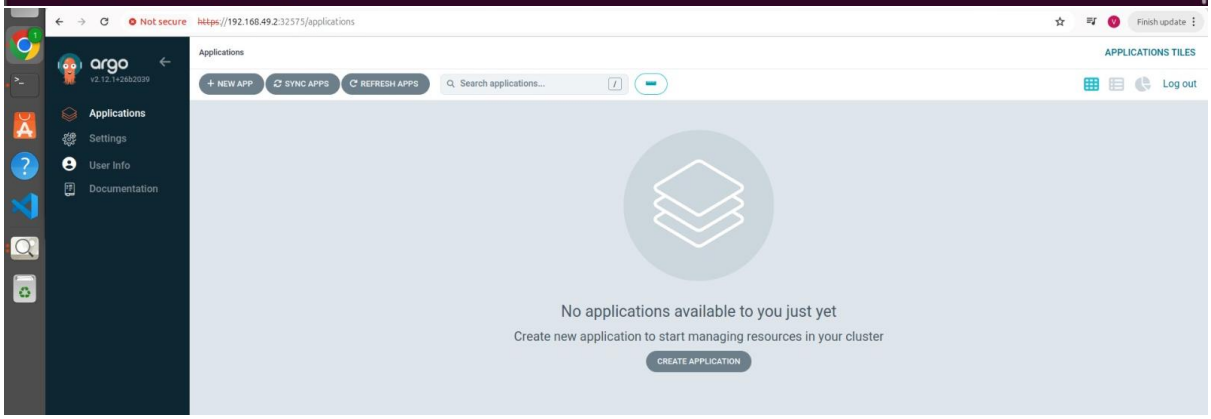
```
varun@DevOps:~$ minikube service list
|-------------|-------------------------------------------------|-------------|--------------------------|
|  NAMESPACE  |                      NAME                       | TARGET PORT |           URL            |
|-------------|-------------------------------------------------|-------------|--------------------------|
| default     | example-argocd-metrics                          | No node port |                          |
| default     | example-argocd-redis                            | No node port |                          |
| default     | example-argocd-repo-server                      | No node port |                          |
| default     | example-argocd-server                           | http/80      | http://192.168.49.2:32575 |
|             |                                                 | https/443    | http://192.168.49.2:31406 |
| default     | example-argocd-server-metrics                   | No node port |                          |
| default     | kubernetes                                      | No node port |                          |
| kube-system | kube-dns                                        | No node port |                          |
| olm         | operatorhubio-catalog                           | No node port |                          |
| olm         | packageserver-service                           | No node port |                          |
| operators   | argocd-operator-controller-manager-metrics-service | No node port |                          |
| operators   | argocd-operator-controller-manager-service      | No node port |                          |
| operators   | argocd-operator-webhook-service                 | No node port |                          |
|-------------|-------------------------------------------------|-------------|--------------------------|
```

Copy the 1st URL and paste in browser.

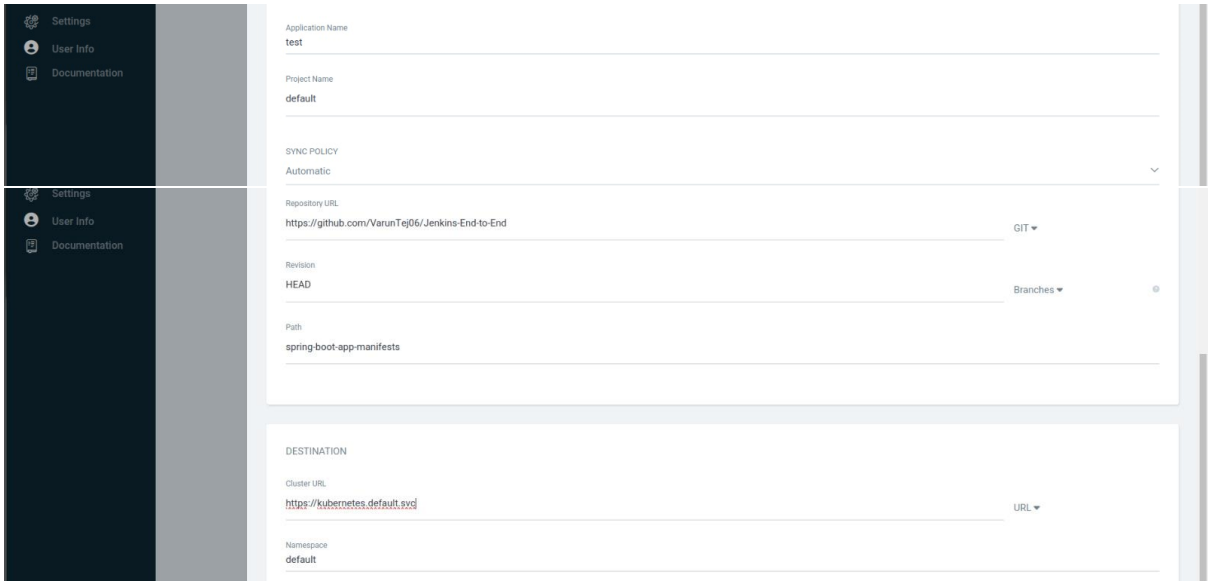


To get password: Copy the Admin password and then change the format of it.

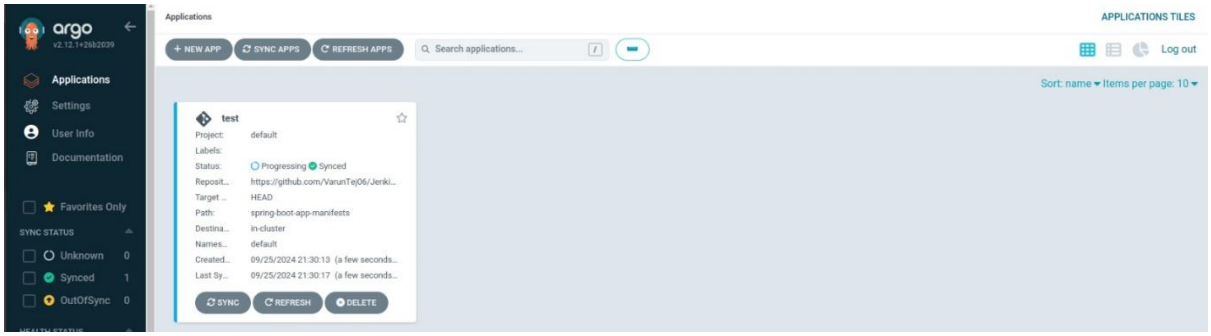
```
varun@DevOps:~$ kubectl get secret
NAME                                TYPE                 DATA   AGE
argocd-secret                       Opaque               5      79m
example-argocd-ca                   kubernetes.io/tls    3      79m
example-argocd-cluster              Opaque               1      79m
example-argocd-default-cluster-config   Opaque           4      79m
example-argocd-redis-initial-password   Opaque           2      79m
example-argocd-tls                  kubernetes.io/tls    2      79m
varun@DevOps:~$ kubectl edit secret example-argocd-cluster
Edit cancelled, no changes made.
varun@DevOps:~$ echo TVZOWElZZEt3bXFwZXpHMFFiakQ0dm8zNmg1RW5BdUg= | base64 -d
MVNXIYdKwmqpezG8QbjD4vo36h5EnAuHvarun@DevOps:~$
```

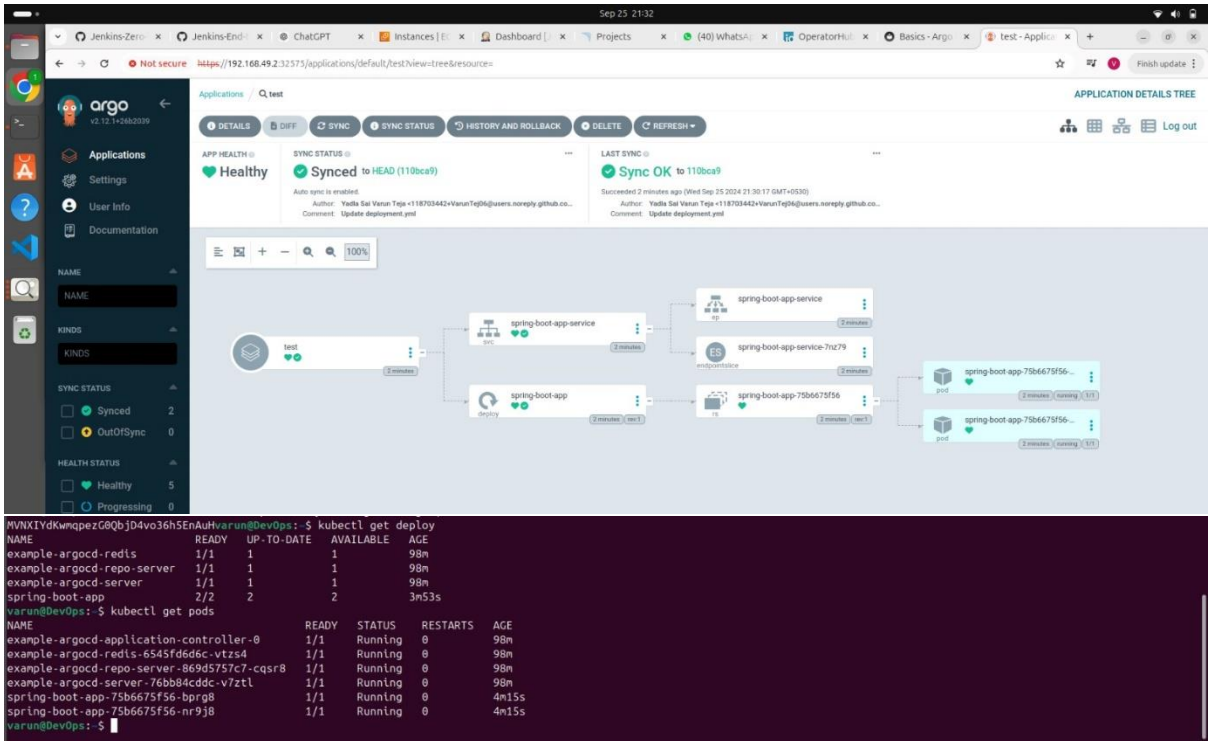


Click on New App:



Click on Create:





Deployment is done, Pods are running. We gave rc count 2 so, 2 pods are running.

CICD is done.

Let's try to do some change in deployment

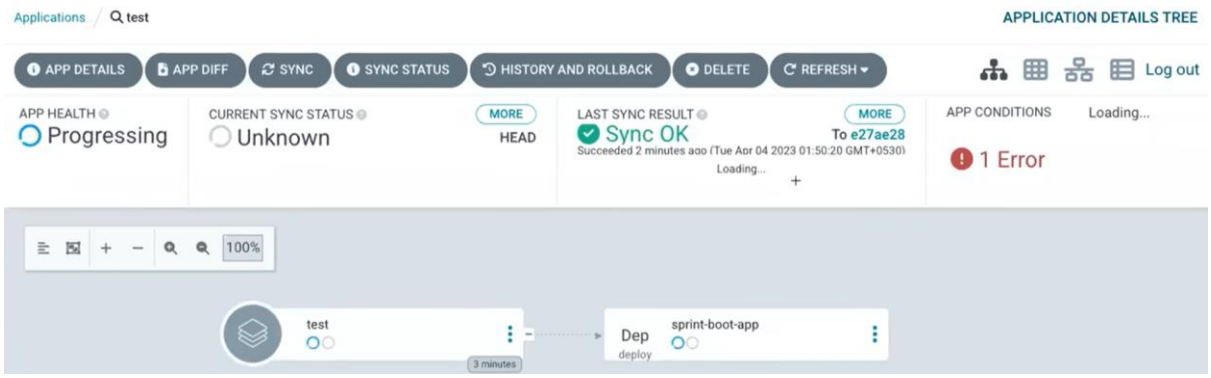
```
spring-boot-app git:(main) ✗ kubectl edit deploy sprint-boot-app
```

Changed the version of image to 2 from 1. Version is not builded yet.

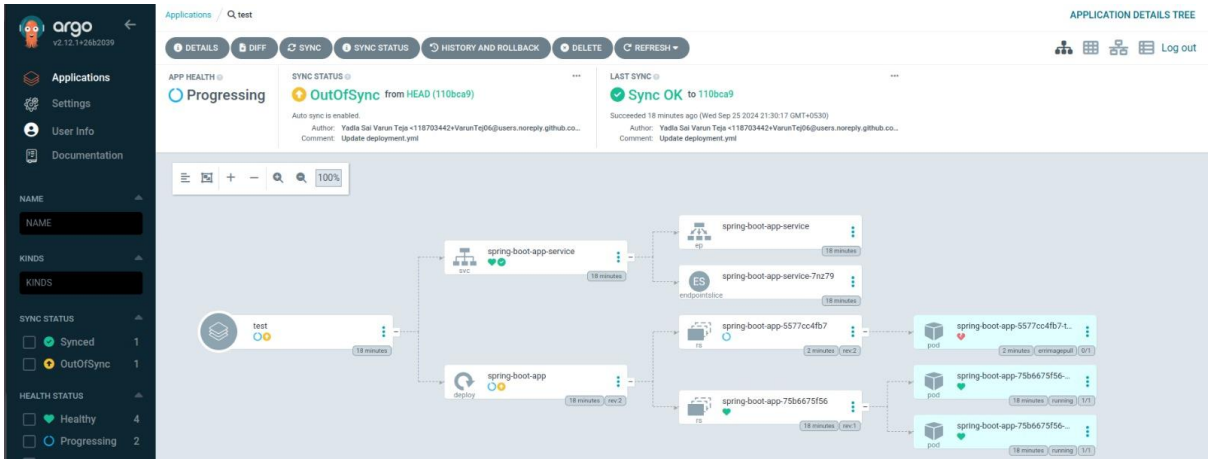
```
spec:
  containers:
  - image: abhishekf5/ultimate-cicd:2
    imagePullPolicy: IfNotPresent
    name: sprint-boot-app
    ports:
```

So, the deployment should fail. Referesh the AgroCD page.

ArgoCD starts updating deployment.



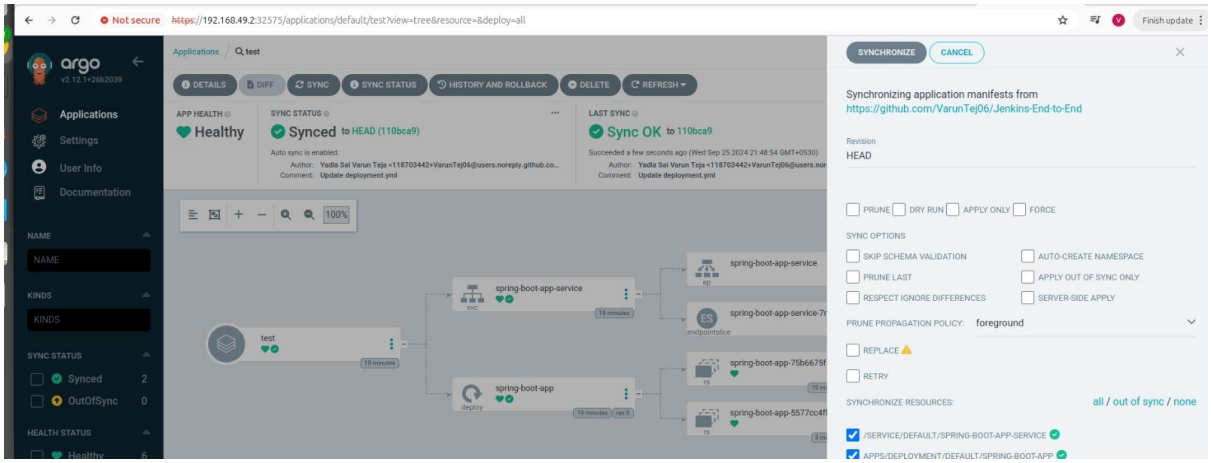
This shows application is out of sync.

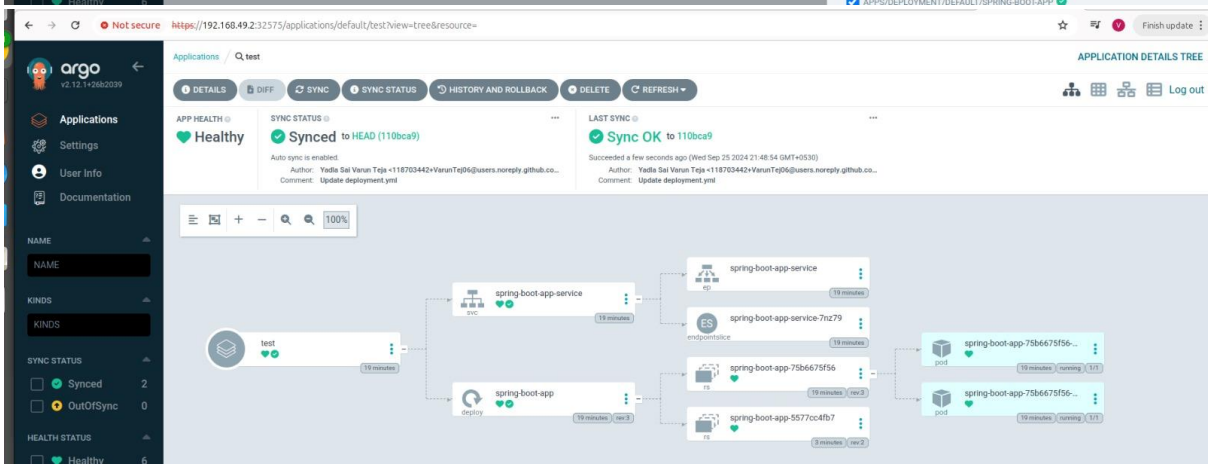


Click on test section to see the differences:

```
130        app: sprint-boot-app                          130        app: sprint-boot-app
131      spec:                                           131      spec:
132        containers:                                   132        containers:
133          - image: 'abhishekf5/ultimate-cicd:2'       133          - image: 'abhishekf5/ultimate-cicd:1'
134            imagePullPolicy: IfNotPresent             134            imagePullPolicy: IfNotPresent
135            name: sprint-boot-app                     135            name: sprint-boot-app
136            ports:                                    136            ports:
```

Click on Sync and then Synchronize:





It gets back to normal.