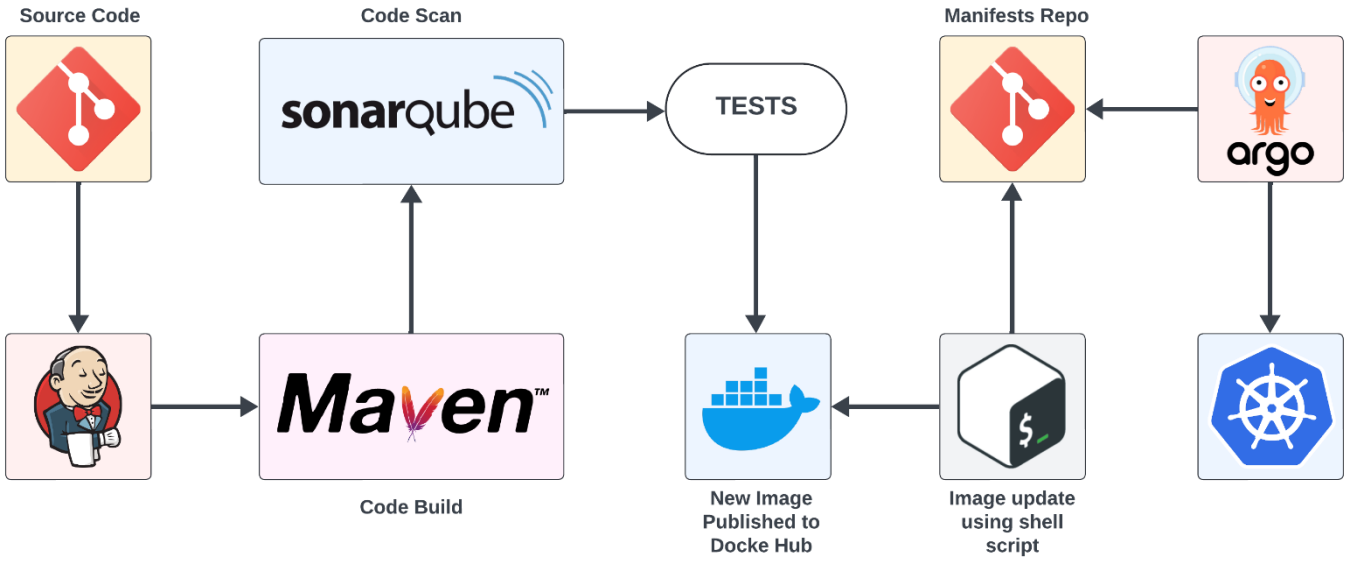
END TO END CICD IMPLEMENTATION

OF SPRING BOOT APPLICATION ON AWS



1. Introduction

The CI/CD pipeline automates the process of building, testing, and deploying a Spring Boot application. By integrating tools like Jenkins, SonarQube, Maven, Docker, Argo CD, and AWS, the pipeline ensures code quality, quick builds, and seamless deployments to a Kubernetes cluster.

2. Tools and Technologies Used:

- **Jenkins**: For continuous integration and automation of the build process.
- **SonarQube**: For static code analysis and ensuring code quality.
- **Maven**: For managing dependencies and building the Spring Boot application.
- **Docker**: For containerizing the application.
- **Argo CD**: For continuous deployment to a Kubernetes cluster.
- **Git**: For version control and source code management.
- **AWS**: For provisioning servers (EC2 instances) to host Jenkins, SonarQube, and other services.

All The codes are available in

- https://github.com/shehan404/CICD-project-1.git
- https://github.com/shehan404/CICD-Project-1-Manifest.git

2. Pipeline Architecture

The pipeline is designed to automate the process from code commits to deployment. Below is the architecture:

Source Code Repository (Git):
The pipeline starts with a source code commit that is manually triggered or scheduled within Jenkins.

Jenkins:
Hosted on an AWS EC2 instance. Jenkins starts the pipeline based on manual triggers or schedules. Jenkins triggers SonarQube to perform a code scan. After a successful code scan, Jenkins initiates the build process using Maven.

SonarQube:
Hosted on an AWS EC2 instance. Analyzes the source code for bugs, vulnerabilities, and code smells. Generates a report that is used to decide whether to proceed with the build.

Maven:
Builds the Spring Boot application. Generates an artifact that is then containerized using Docker.

Docker:
The build artifact is containerized. A new Docker image is created and pushed to Docker Hub.

Manifest Repository (Git):
A separate repository for Kubernetes manifests. The manifest files are updated with the new Docker image using a shell script.

ArgoCD:
ArgoCD monitors the manifests repository. When changes are detected, ArgoCD automatically deploys the updated application to the Kubernetes cluster hosted on AWS.

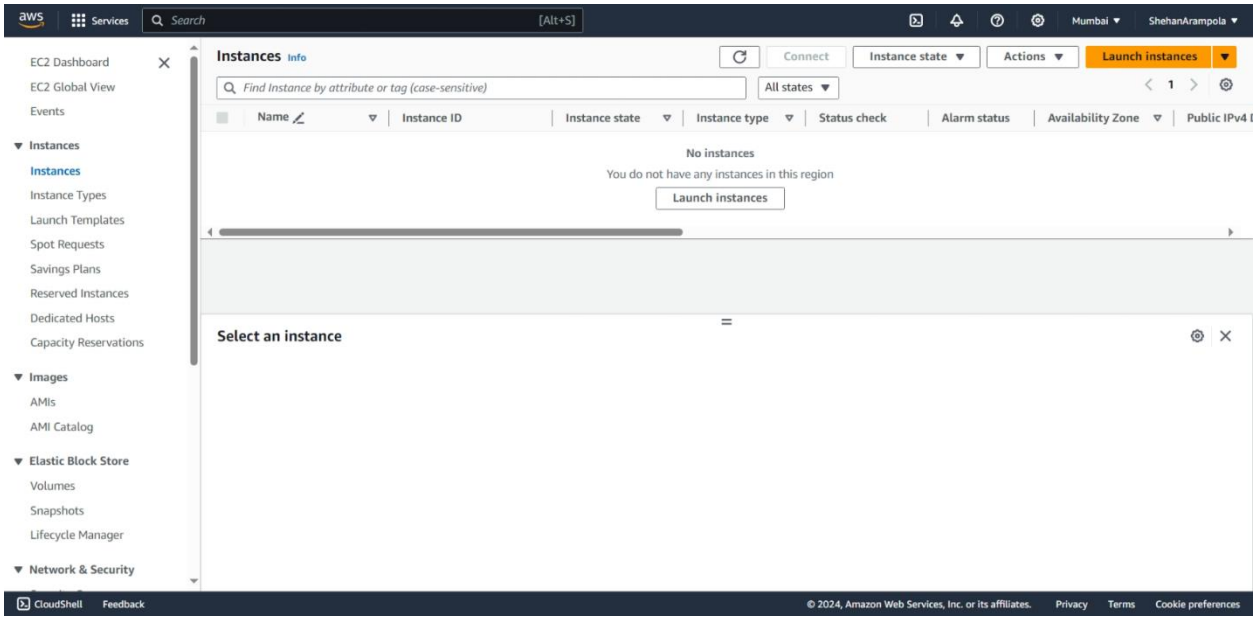
3. Setup and Configuration

1. **AWS**:

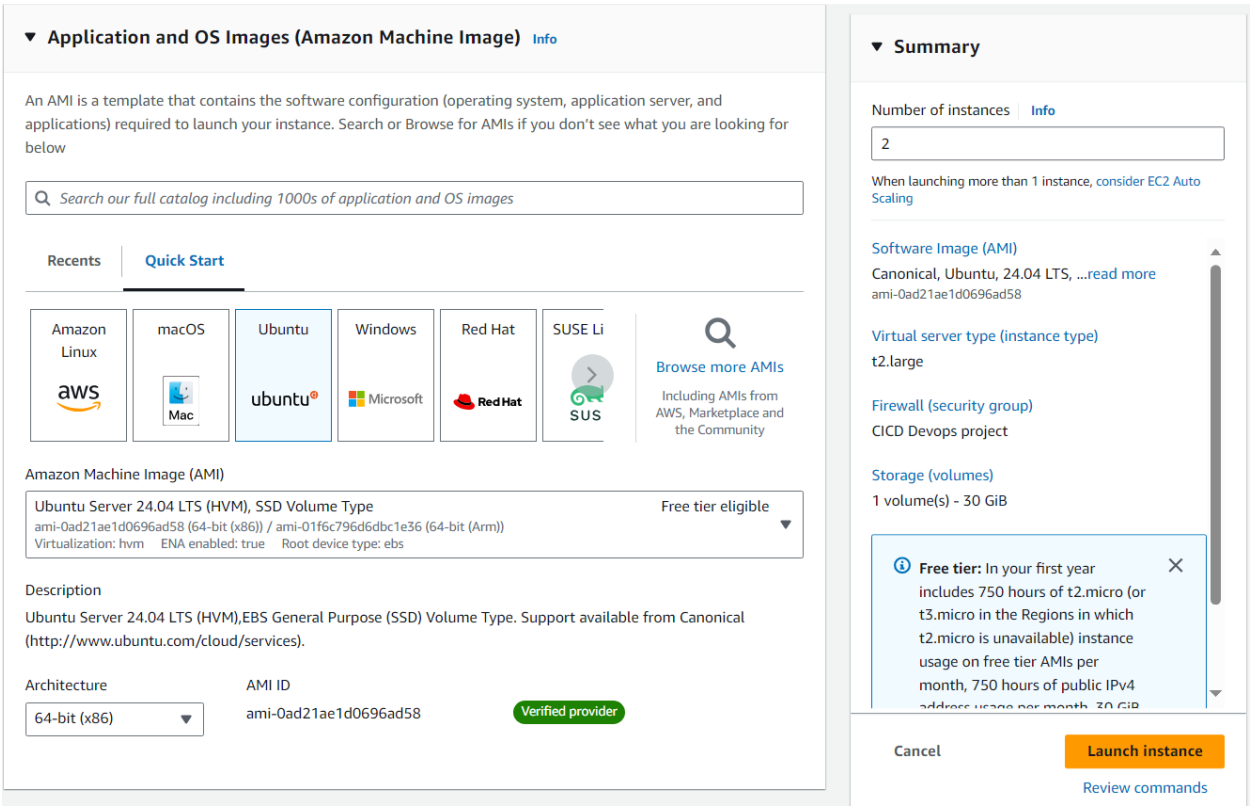
Provision EC2 instances. Follow these steps.

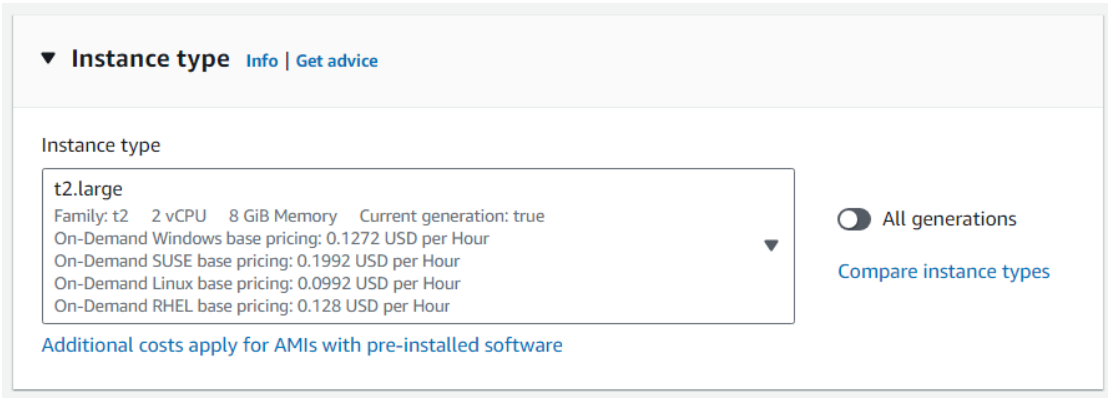
Go to AWS console and create two EC2 instance as follows with the same settings choose the name as you like (Mine: Jenkins-server, Deploy-server).

Click Launch instances

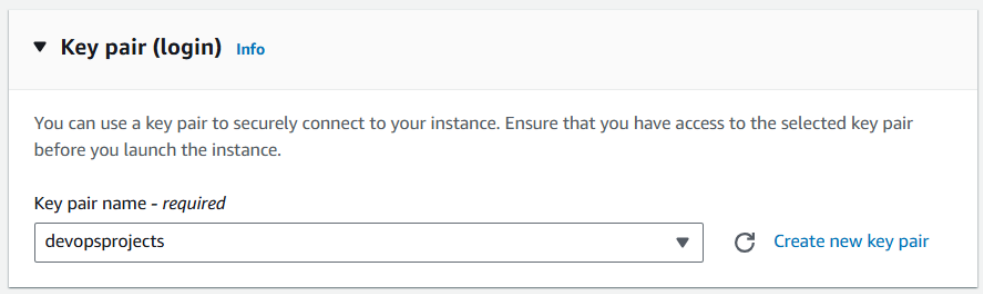


Setup following settings

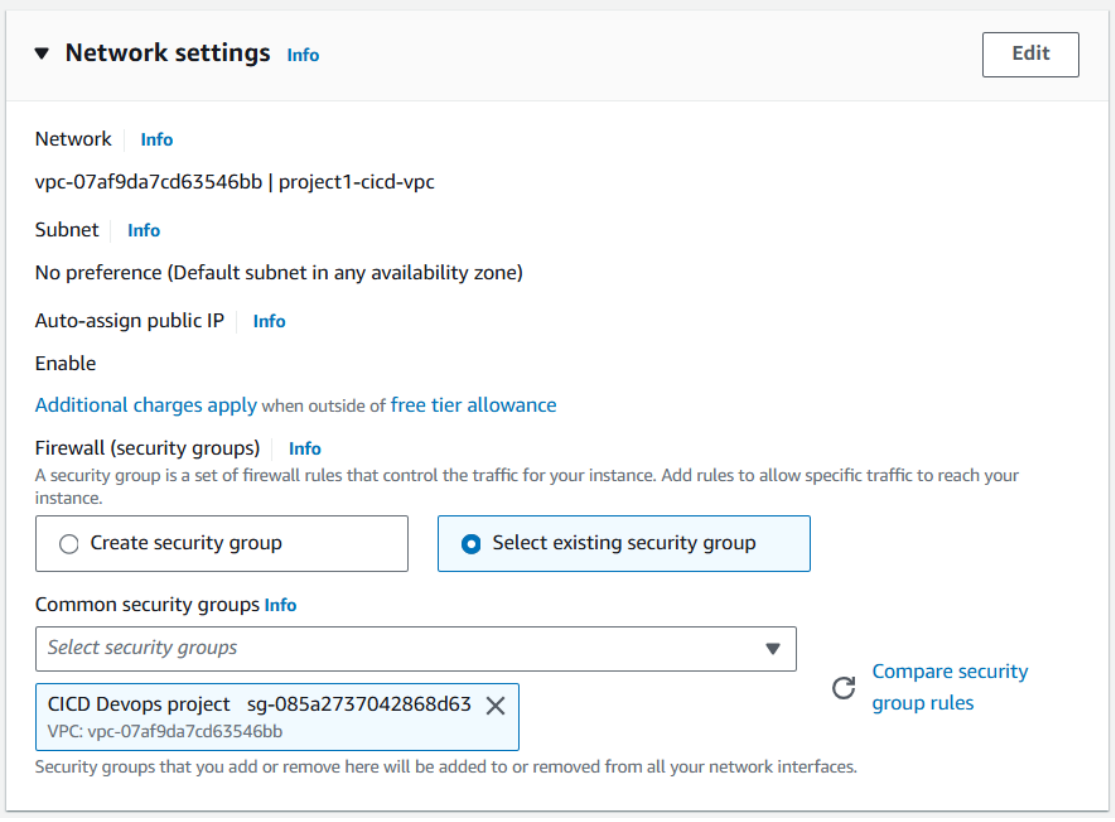


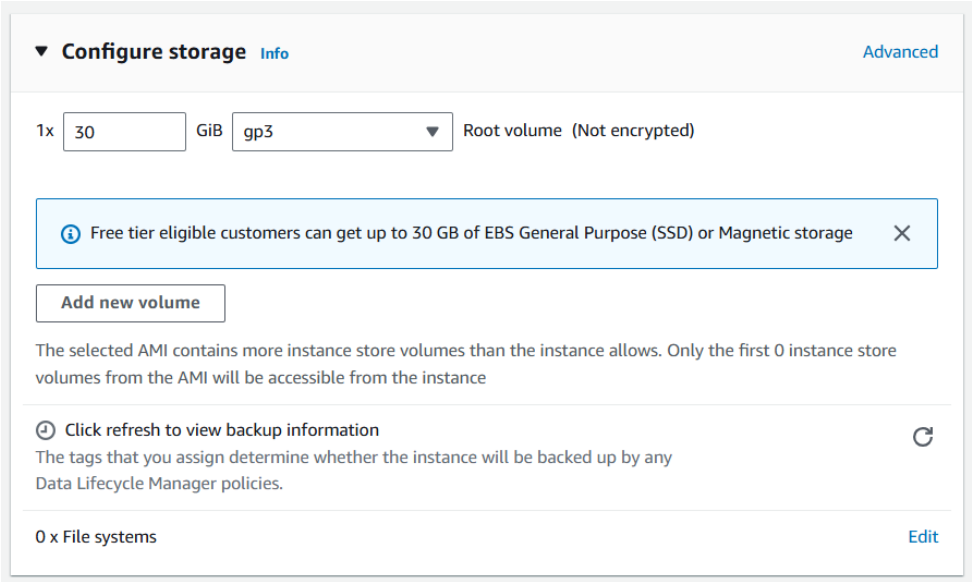


Choose key pair. If you don’t have one create a new key pair. The .pem file created is used for authenticating when remote login into EC2 instances.

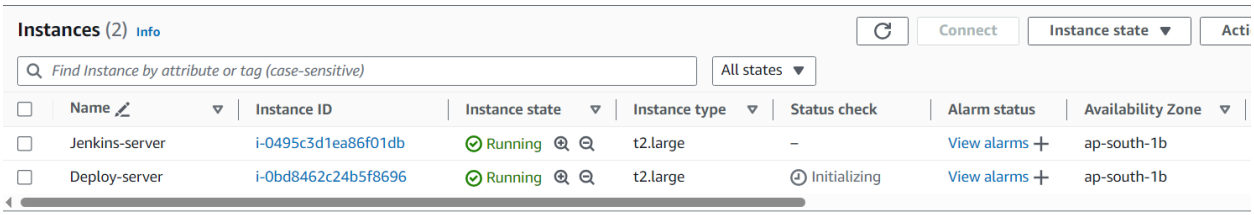


Choose a security group. In the inbound rules of SG, we allow all inbound traffic to make our project easy. But in real situations it should be allowed only traffic that we needed. You can simply search for security group in search bar and setup it.



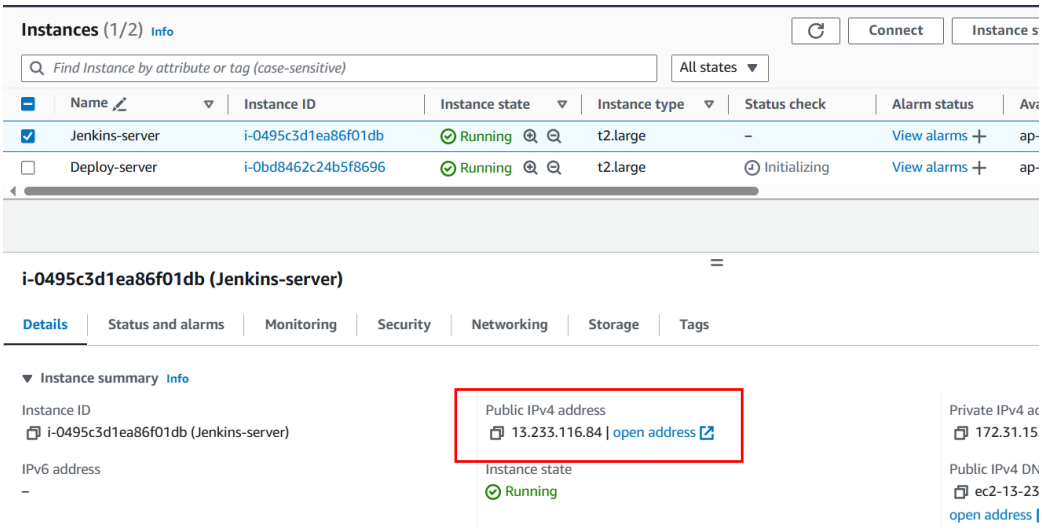


Click Launch instance and instances will be created. Now come to instances menu and rename them.

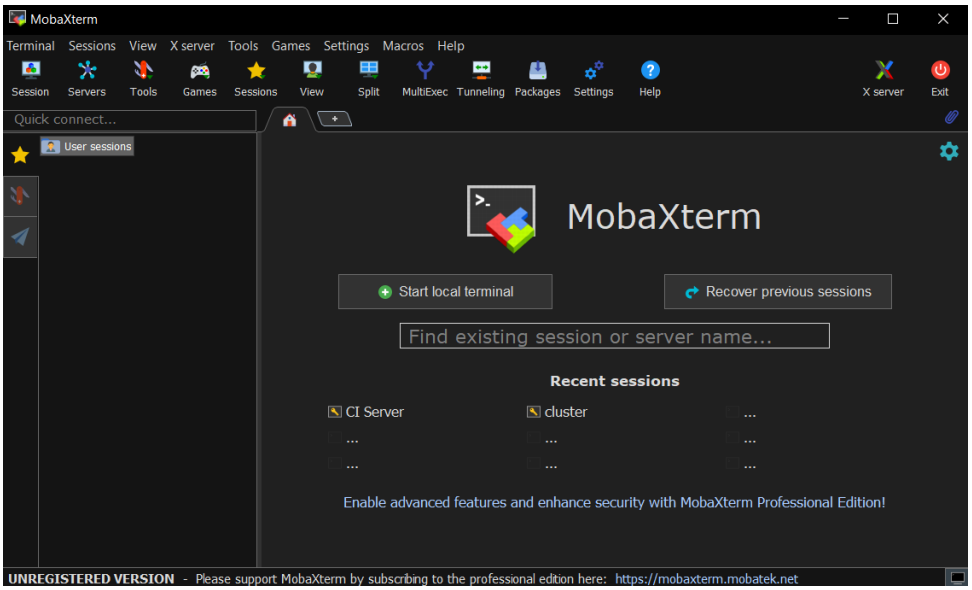


Now instances are created.

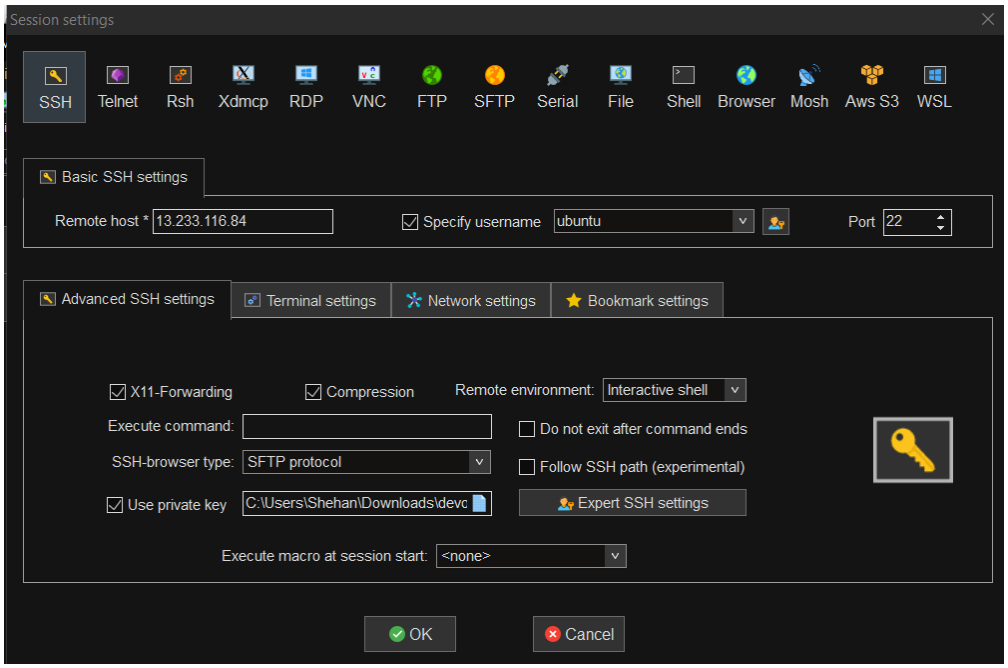
Select each instance and copy the public IP address of both instances.

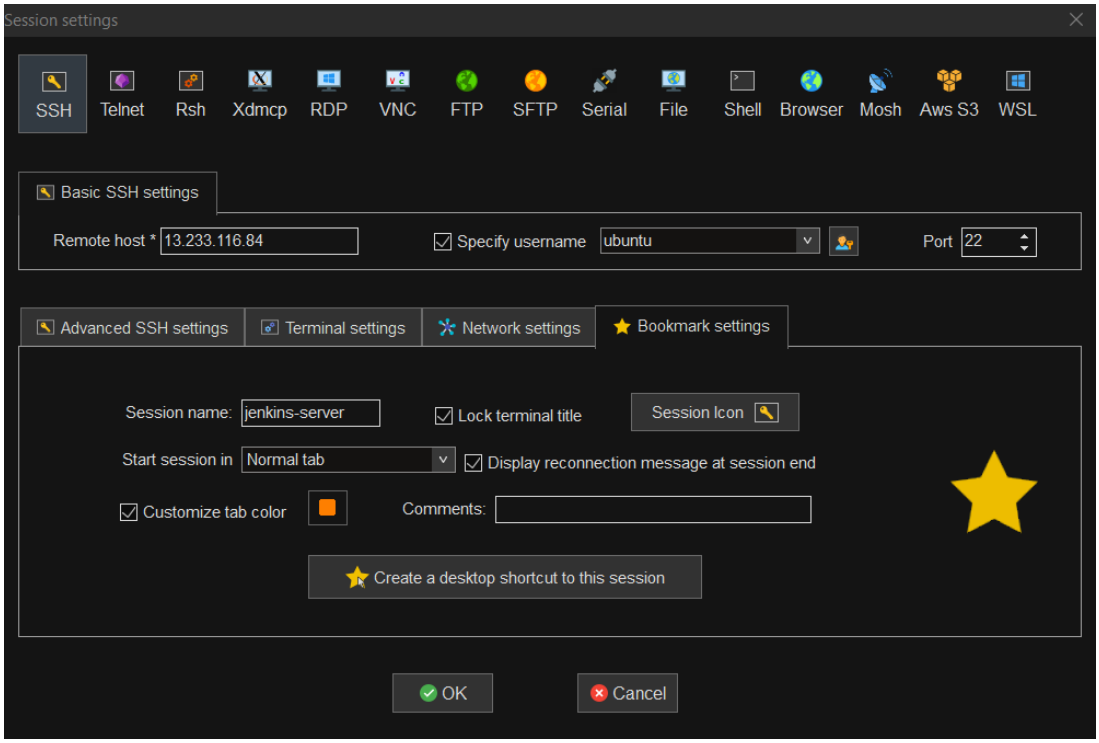


Let’s get remote access to EC2 instances using MobaXterm. Download and install the MobaXterm.

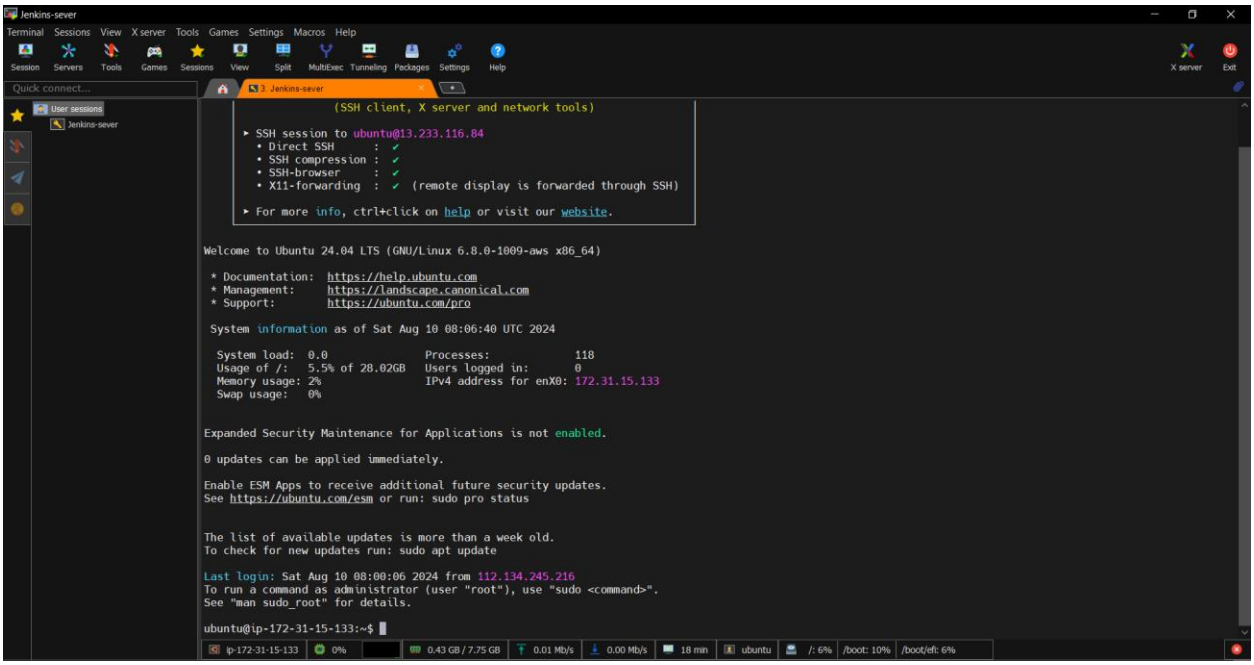


Go to session > SSH and enter the Public IP Address of EC2 instance and specify username as ubuntu in Basic SSH settings. In advanced settings, select use private key and brows for .pem file I mentioned previously. Click OK. In Bookmark settings set session name as Jenkins-server and choose a color for easy identification.

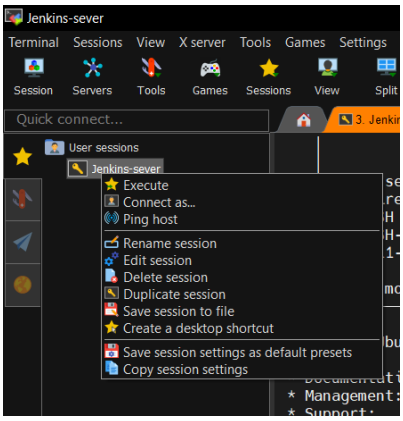




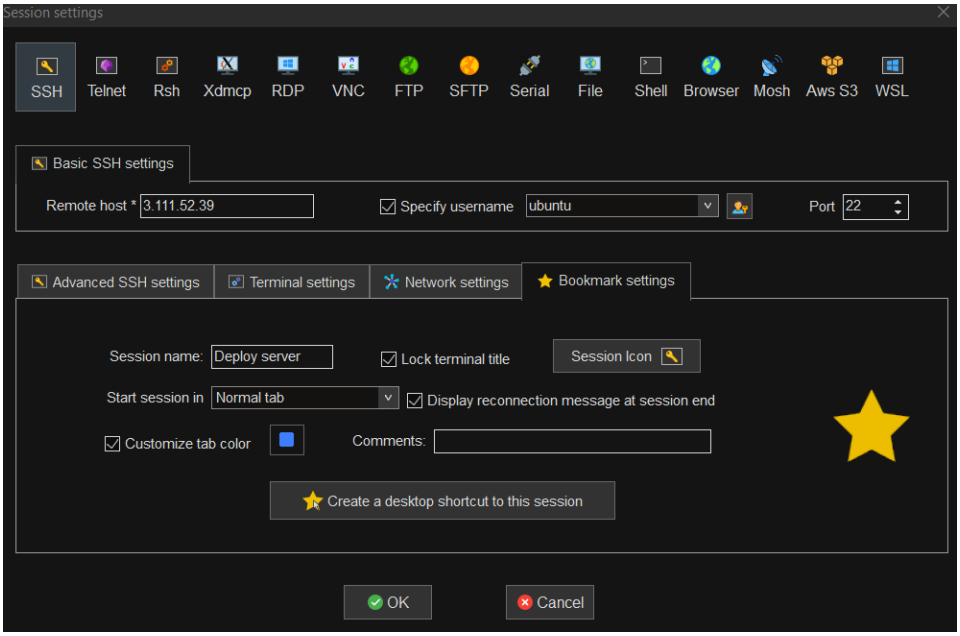
Great now you are remotely connected to the one EC2 instance. In my case I have connected to Jenkins-server.



Right click on Jenkins-server and Click Duplicate Session

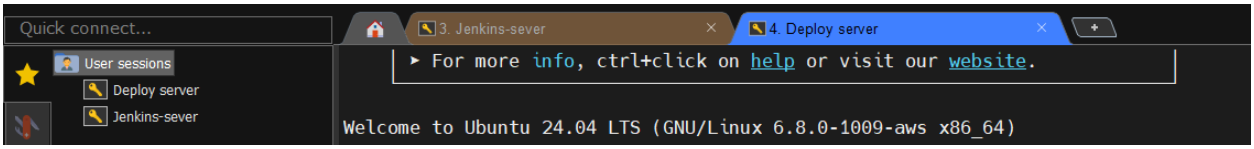


Now right click on duplicated session and click rename session. Insert IP address of other EC2 instance and change the session name and color. Click ok.



Run `sudo apt update` in both servers.

Great We have setup or servers 😊



2. Jenkins

Go to Jenkins server and install jdk. Because Jenkins requires java.

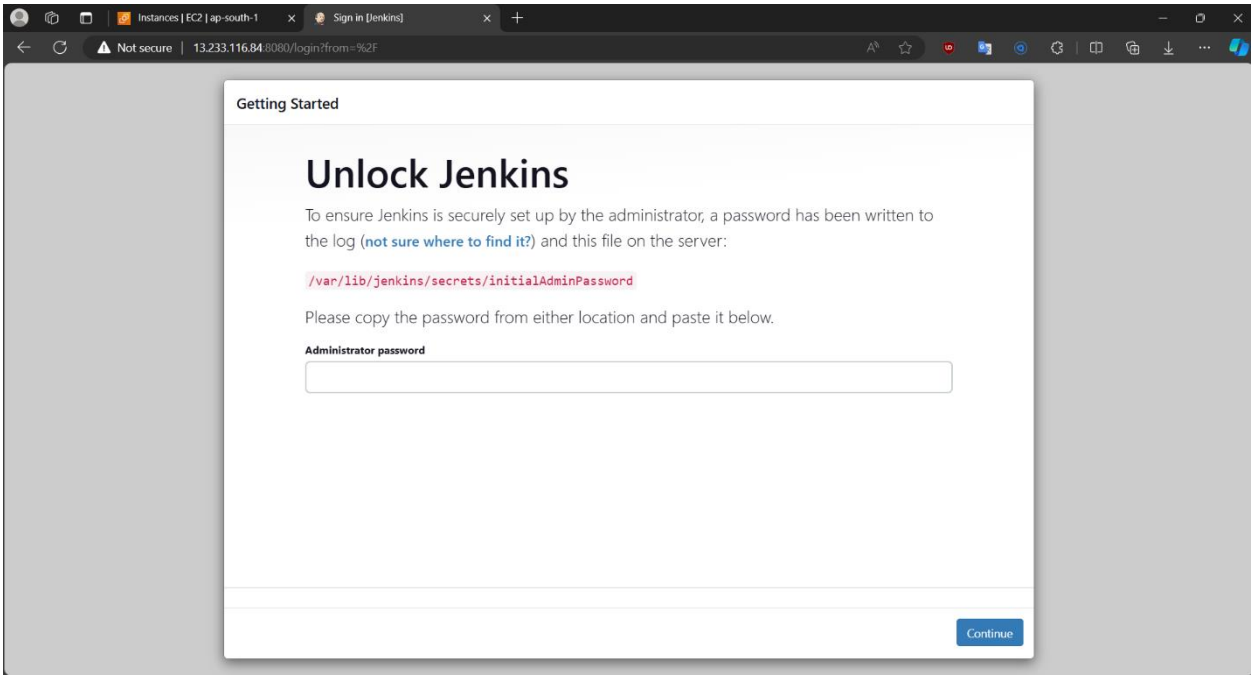
```
sudo apt install default-jre
```

Now install Jenkins using the following commands. (It is much easier if insert following commands in .sh file and run it)

```
sudo wget -O /usr/share/keyrings/jenkins-keyring.asc \
  https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key
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
sudo apt-get install Jenkins
```

Jenkins is default run in port 8080. Go to your browser and insert the following link

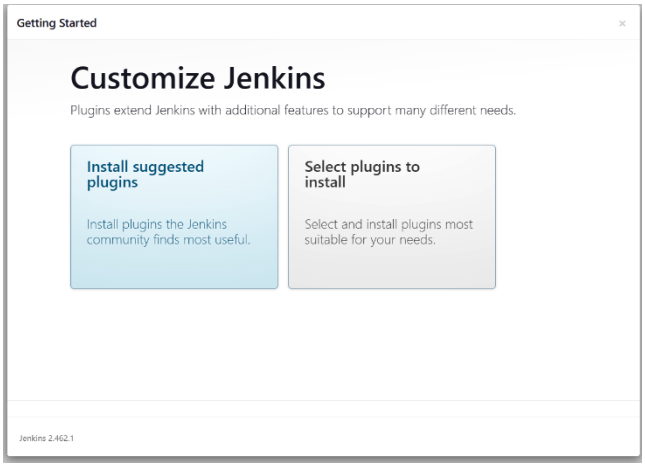
```
http://<jenkins-server ec2 instance public ip address>:8080
```

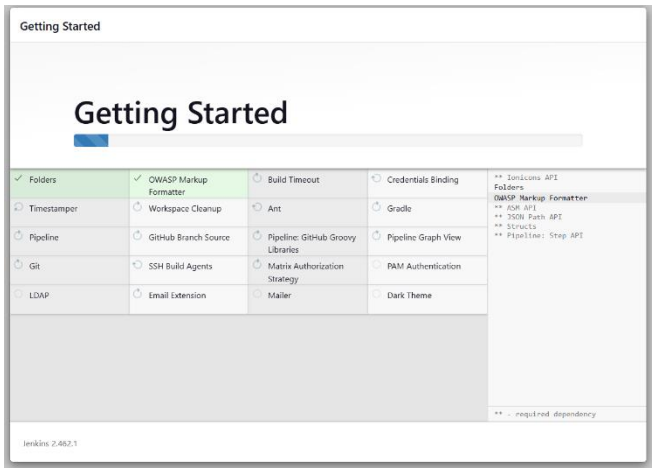


For get the administrator password run following command.

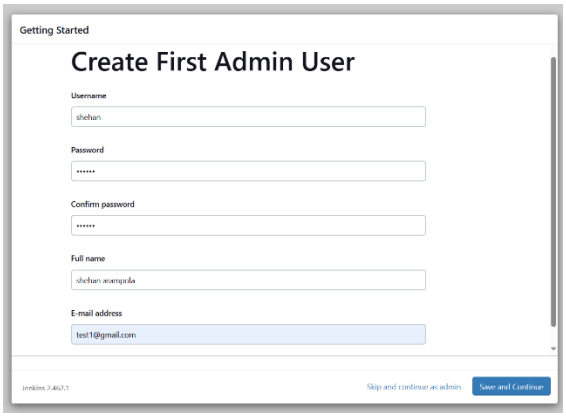
```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

Copy the password given and insert it in the above textbox. In next page choose Install suggested plugins. Jenkins will install the plugins it needed.

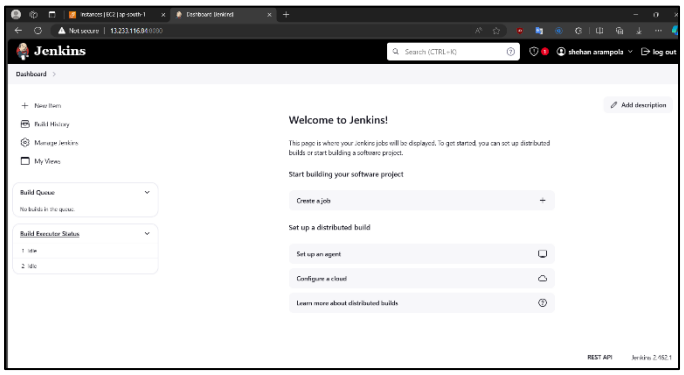




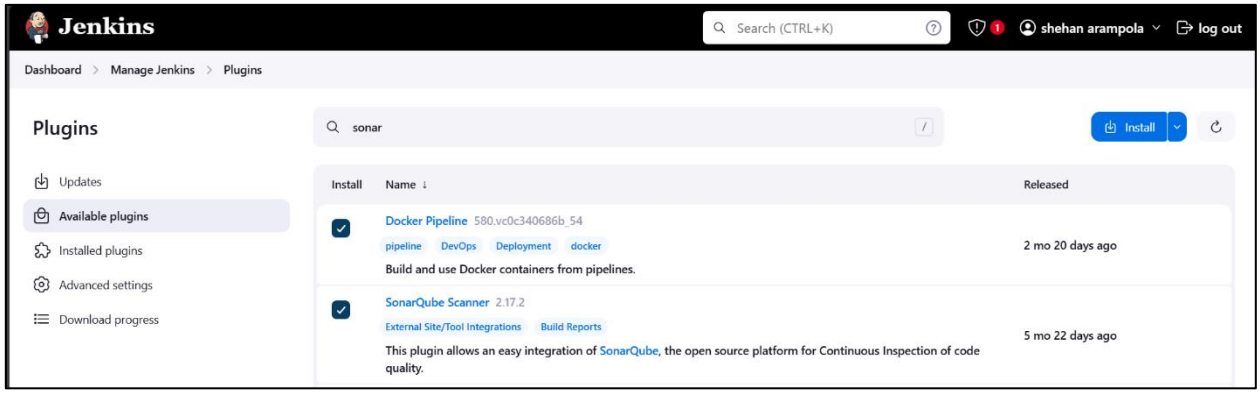
Create username and password



Cool We successfully installed Jenkins 😊



Go to Manage Jenkins > Plugins > Available plugins and search for docker pipeline and SonarQube Scanner and install them.



3. SonarQube

Go to Jenkins-server EC2 on MobaXterm and install docker using following commands. Because we use SonarQube in a docker container.

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
/etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin
docker-compose-plugin
```

Run `sudo docker run hello-world` to confirm the docker installation.

Run `sudo chmod 666 /var/run/docker.sock` to give permission to all users to run docker commands

Now install SonarQube using following command

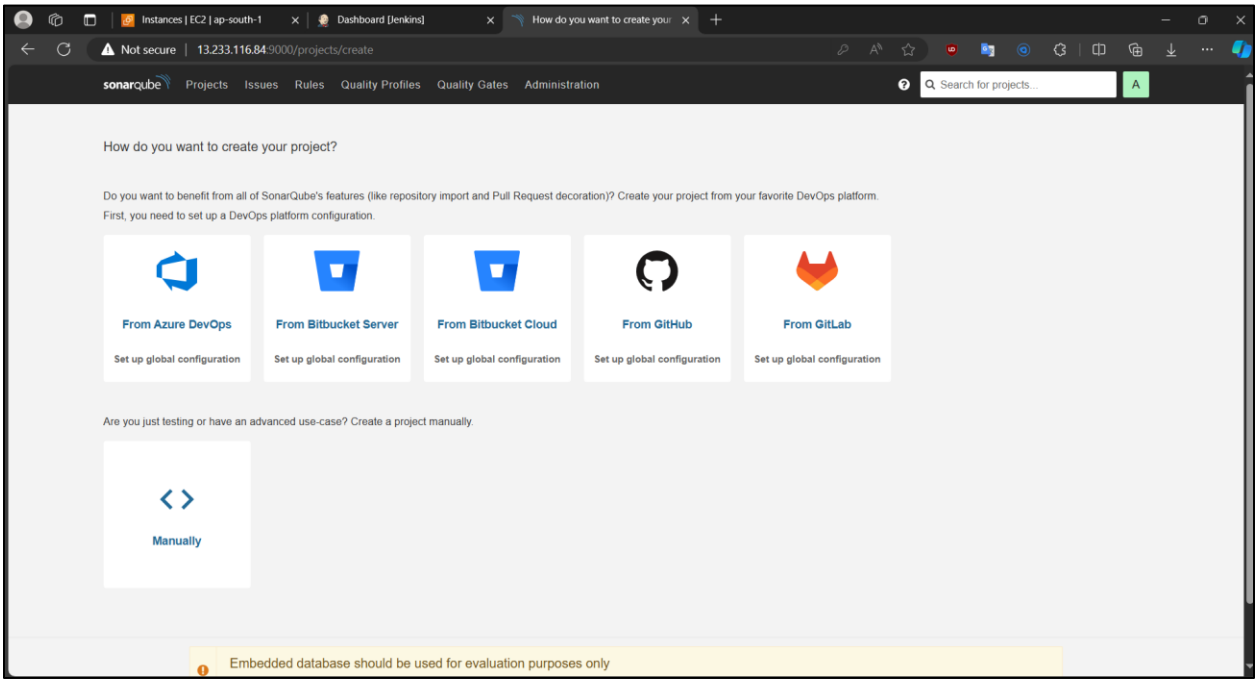
```
docker run -d --name sonar -p 9000:9000 sonarqube:lts-community
```

Cool 😊. Now SonarQube is running on port 9000.

Go to browser and insert following link.

```
http://<jenkins-server ec2 instance public ip address>:9000
```

Login to SonarQube. Default username: admin, Password: admin. Login and create a new password.



Great 😊.

4. Setup credentials

First, Let’s configure credentials for different platforms which we need in upcoming steps.

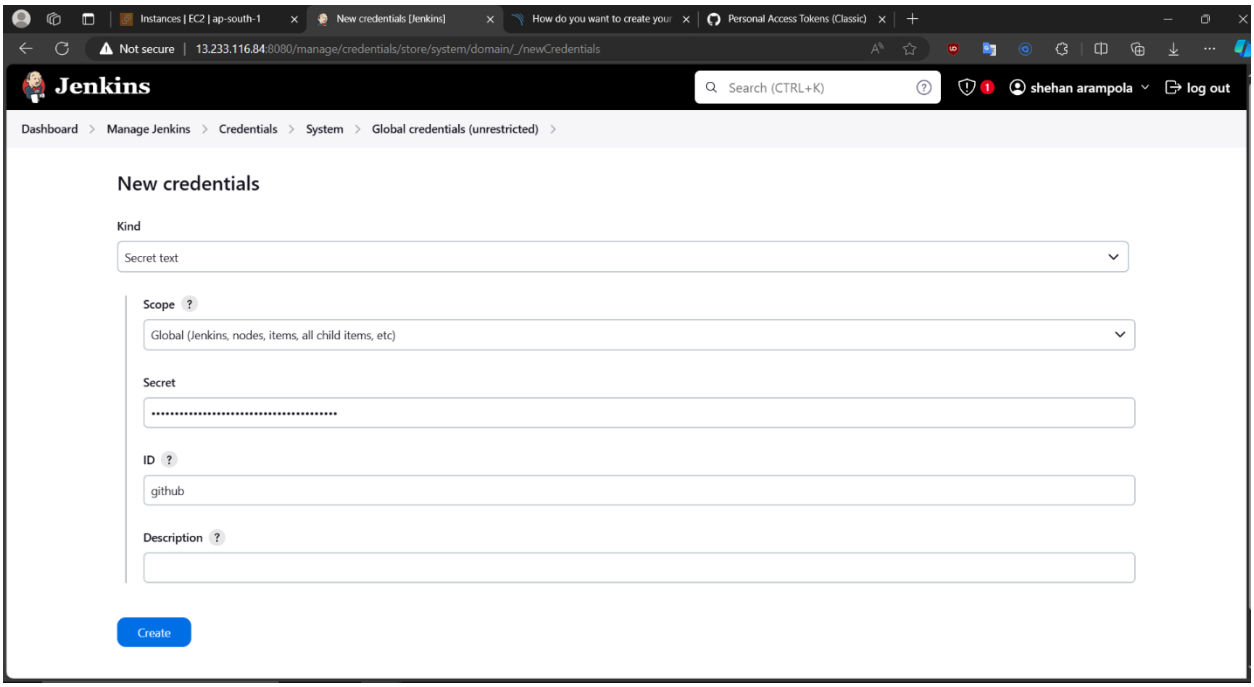
Go to Jenkins Dashboard. Manage Jenkins > Credentials. Click on global. Click on Add Credentials.

1. GitHub (If you use private repo)

Go to GitHub and generate a token in Settings > Developer Settings > Personal access tokens > Token (classic). Give required permissions and generate a token. Save it.

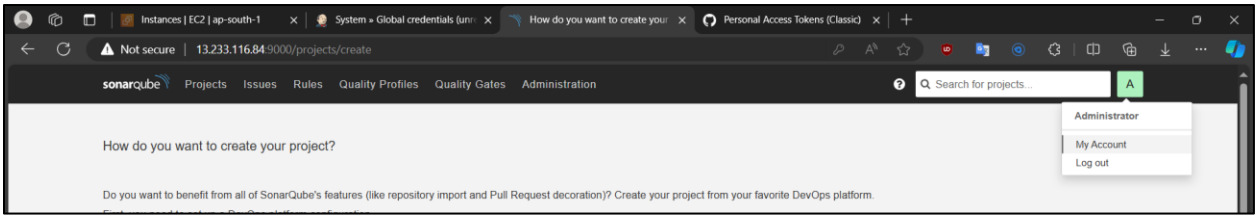
Now in Jenkins under Add Credentials give that token as follows:

Paste the token in Secret. Click create.

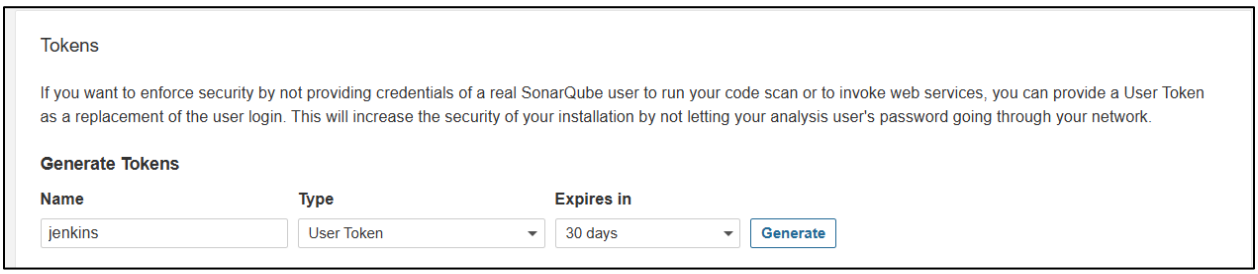


2. SonarQube

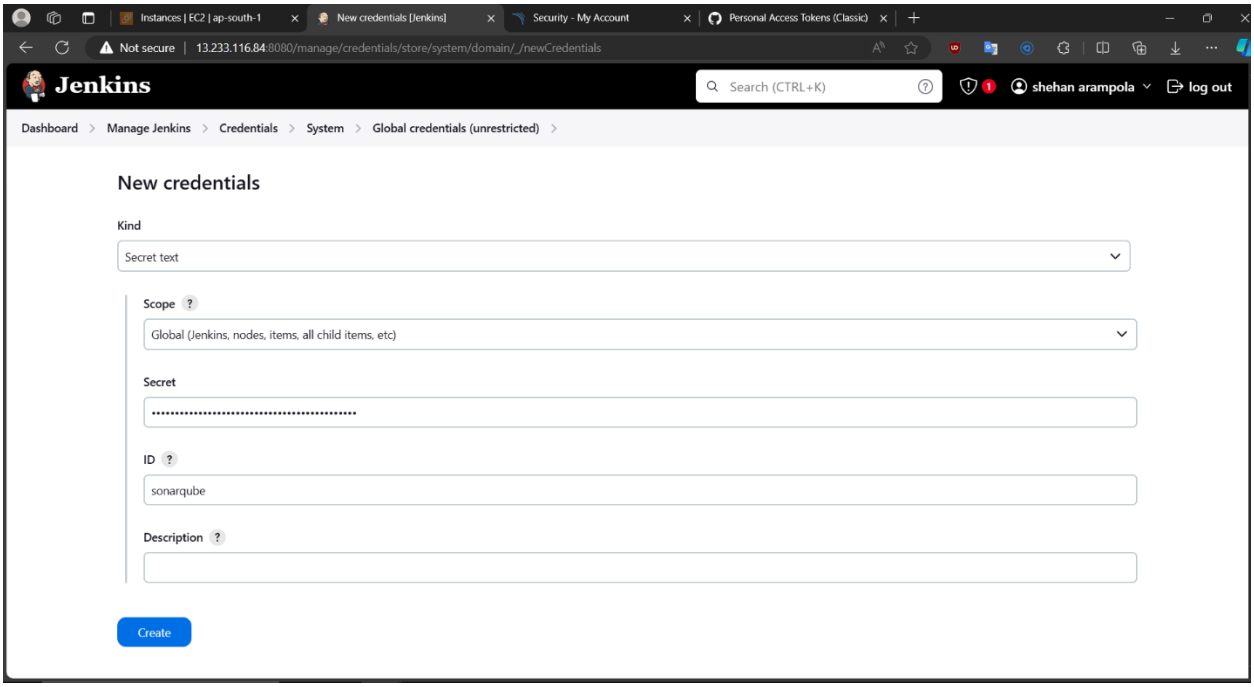
Go to SonarQube page and click on Profile > My account



Go to security and generate a token as follows

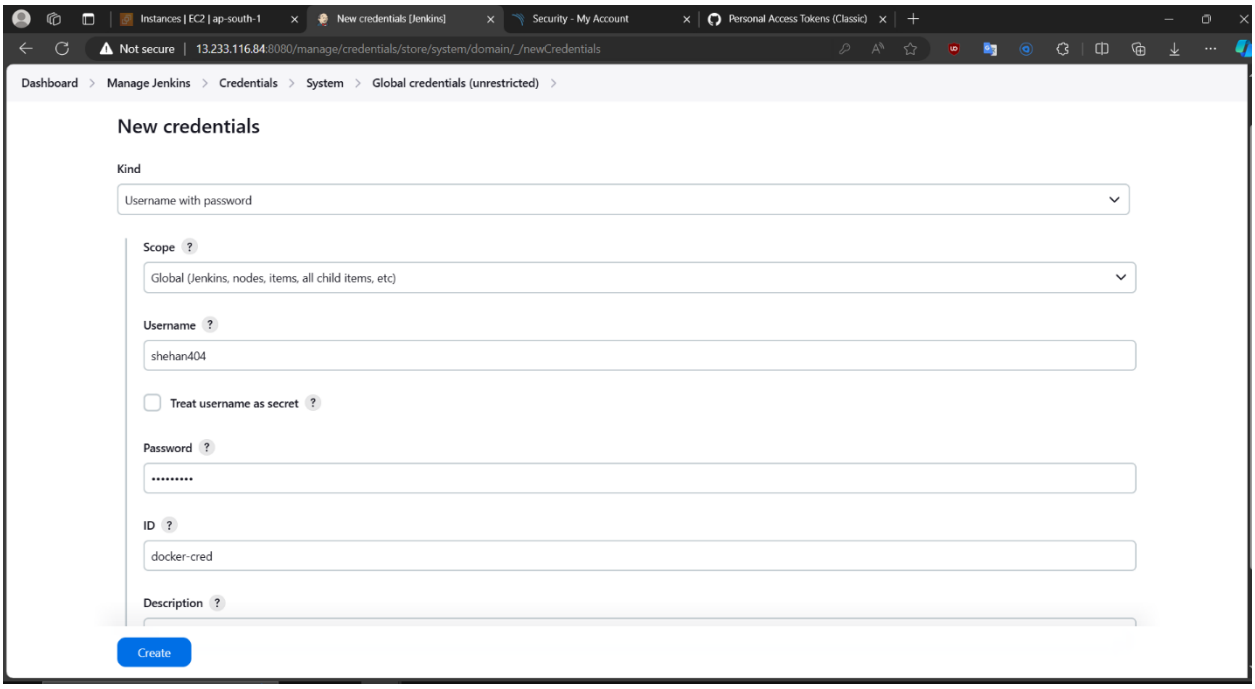


Now copy the token generated and go to Jenkins. Add credentials. Create credentials as follows.

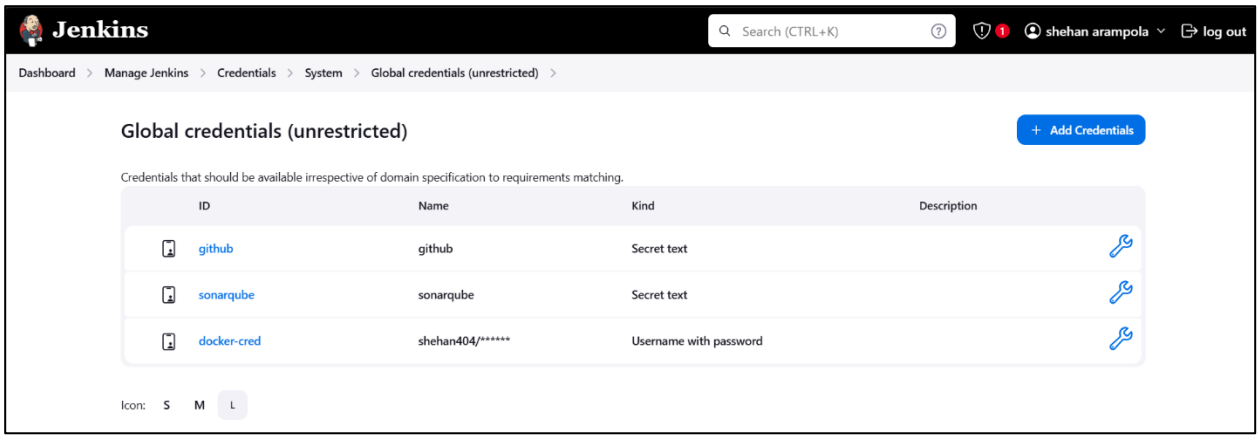


3. Docker

Go to Jenkins add credentials and enter your Docker Hub credentials as follows.



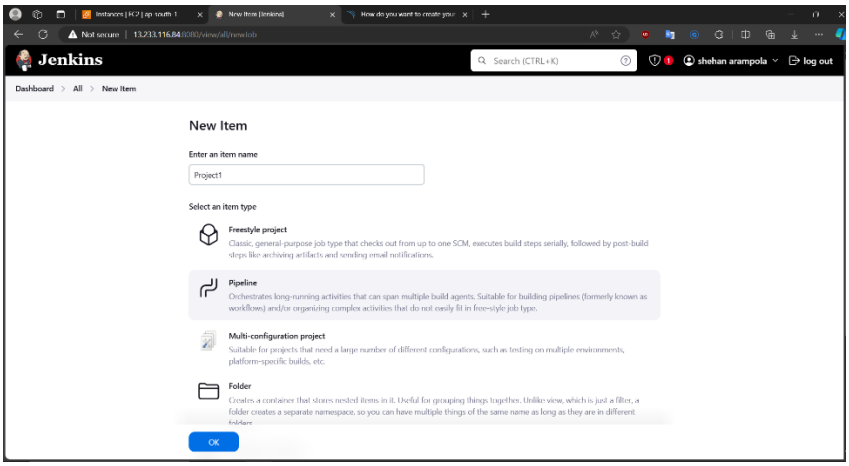
Great Now you have created following credentials



5. CICD Pipeline

We use Jenkins for CI. We use ArgoCD for CD.

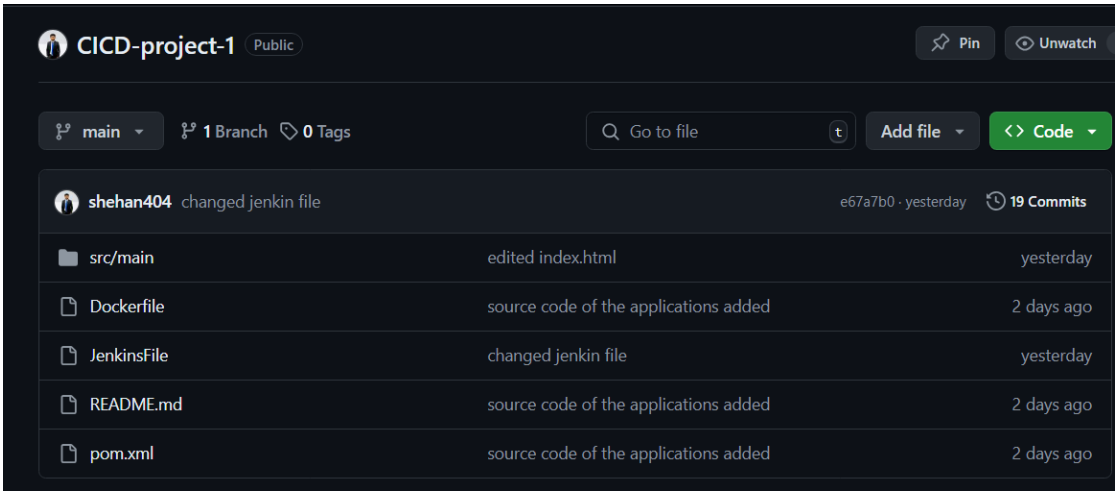
Go to Jenkins Dashboard and click on New Item. Enter a name and select Pipeline. Click Ok.



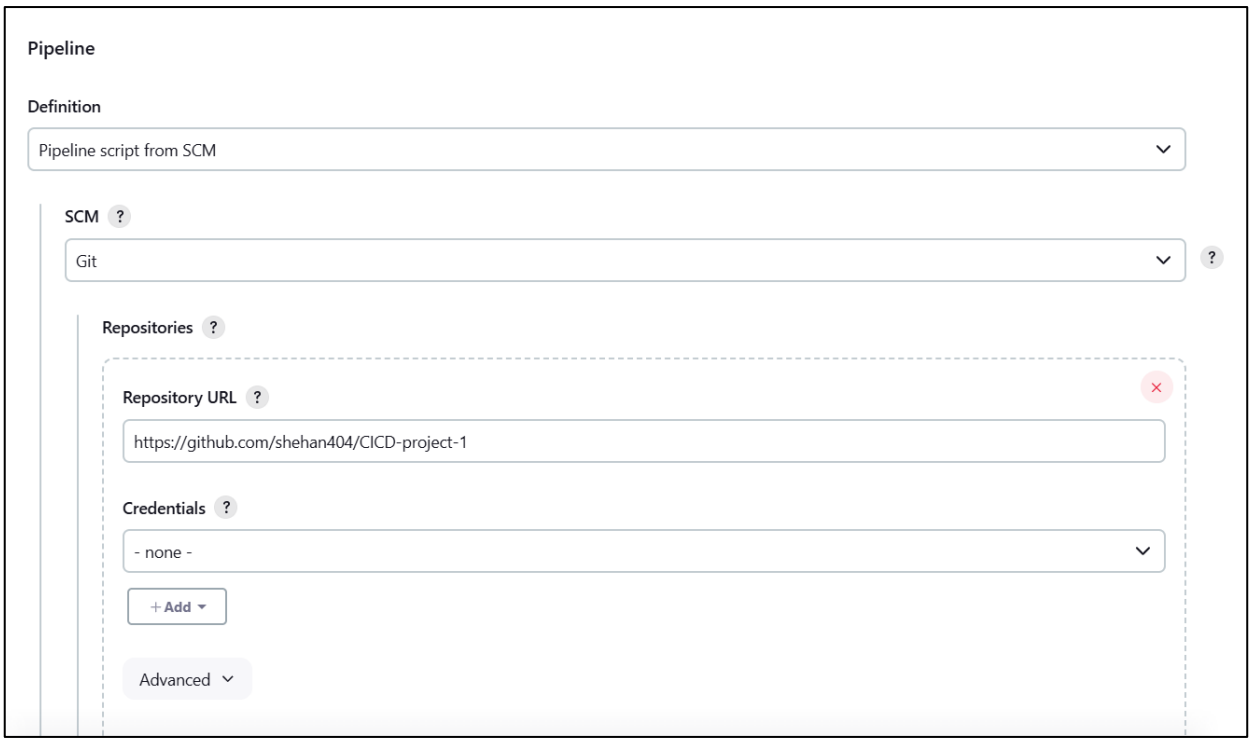
In configuration, under the pipeline choose “Pipeline script from SCM” in definition section. This allows us to get groovy pipeline script from GitHub instead of writing it directly on Jenkins.

I have added application source code and JenkinsFile in the following repository. Clone it into your own repository

https://github.com/shehan404/CICD-project-1.git



Now, in Jenkins after selecting “Pipeline script from SCM” setup it as follows and save.



Before build go to JenkinsFile in the github and do following changes.

1. Under ‘Static Code Analysis’ stage set

```
SONAR_URL = http://<jenkins-server ec2 instance public ip address>:9000
```

In my case http://13.233.116.84:9000/

2. Under 'Build and Push Docker Image' stage set

```
DOCKER_IMAGE = "<dockerhub user>/ultimate-cicd:${BUILD_NUMBER}"
```

In my case

```
DOCKER_IMAGE = "shehan404/ultimate-cicd:${BUILD_NUMBER}"
```

3. Under 'Update Deployment File' stage set

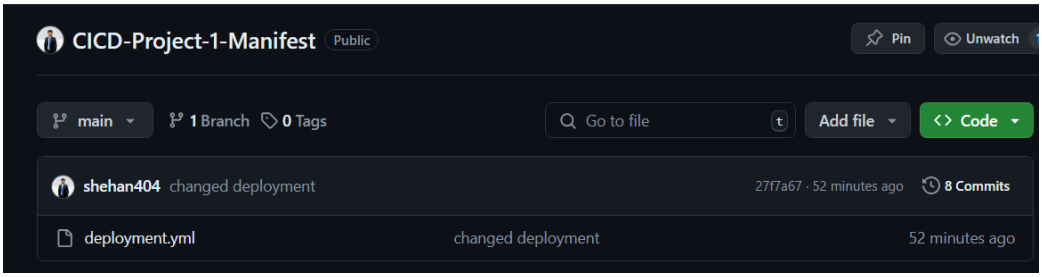
```
GIT_REPO_NAME = "<Manifest repo name>"
GIT_USER_NAME = "<github usename>"
```

In my case

```
GIT_REPO_NAME = "CICD-Project-1-Manifest"
GIT_USER_NAME = "shehan404"
```

I have added manifest repo in following URL. Clone it into your GitHub repo

https://github.com/shehan404/CICD-Project-1-Manifest



Now commit changes.

Great. Now we are ready to Build 😊.

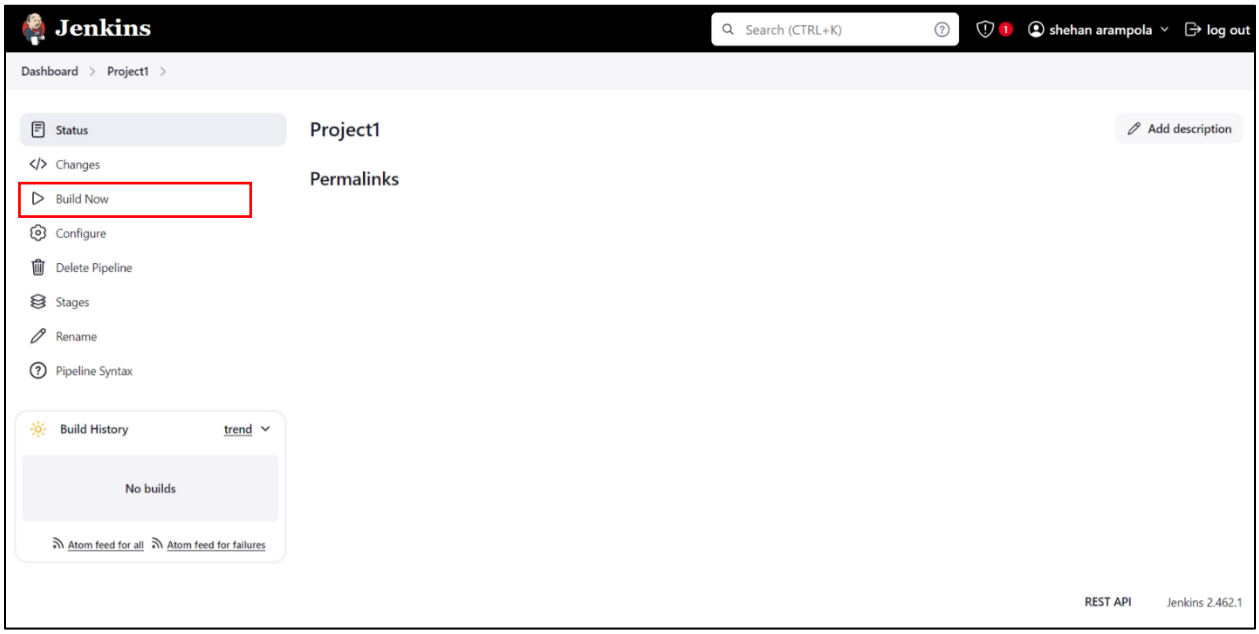
Before build it is a good practice to restart Jenkins.

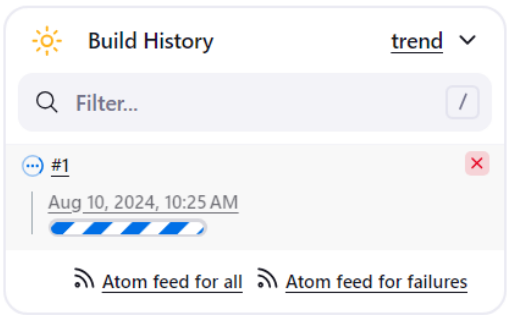
Go to URL and add /restart to it and enter.

In my case http://13.233.116.84:8080/restart

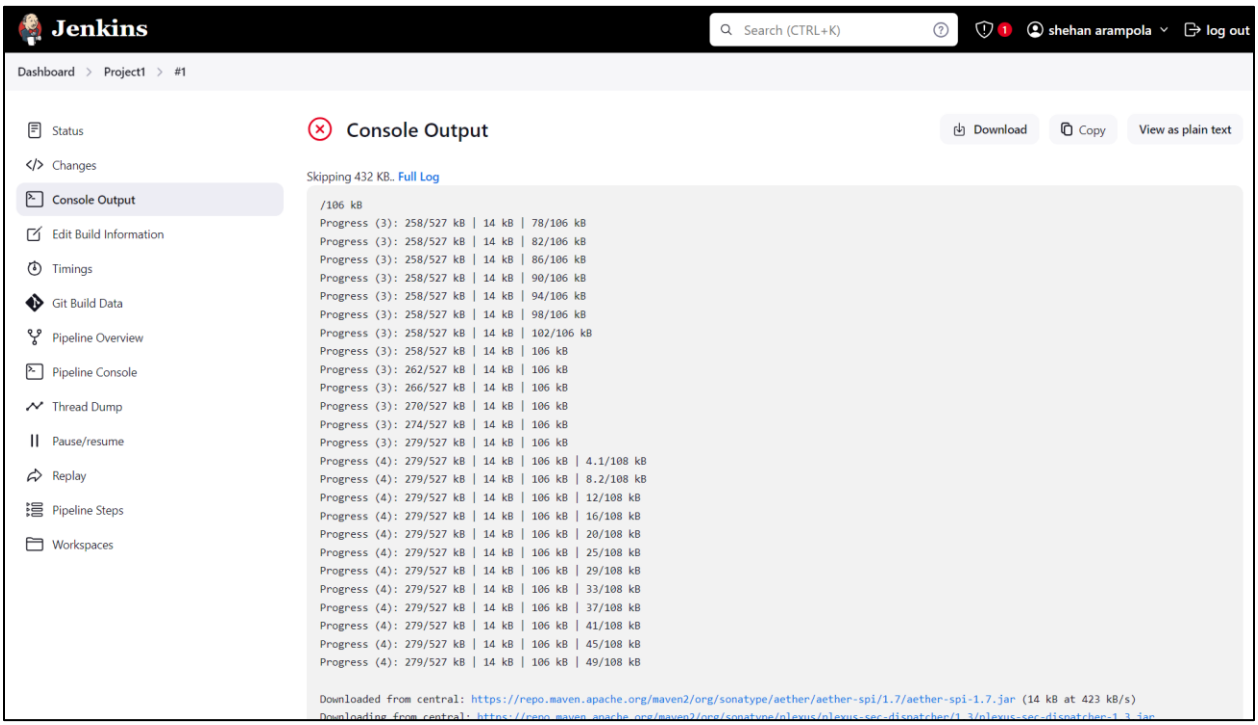
Jenkins will restart. Then login again.

Now Go to Jenkins Dashboard > Project1 and click on Build Now. If everything done correctly no errors will occur





Click on build number and go to console output to find if there any error occurred.

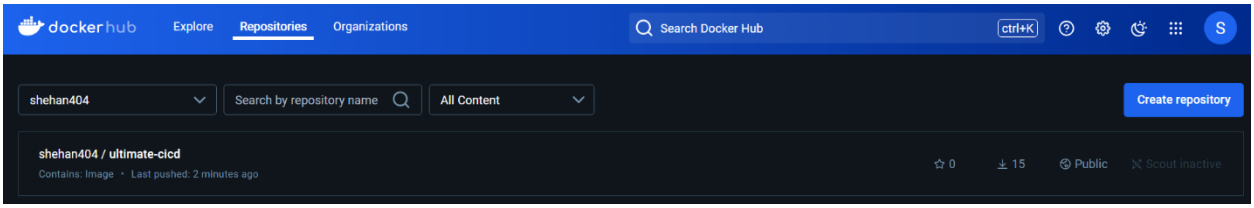


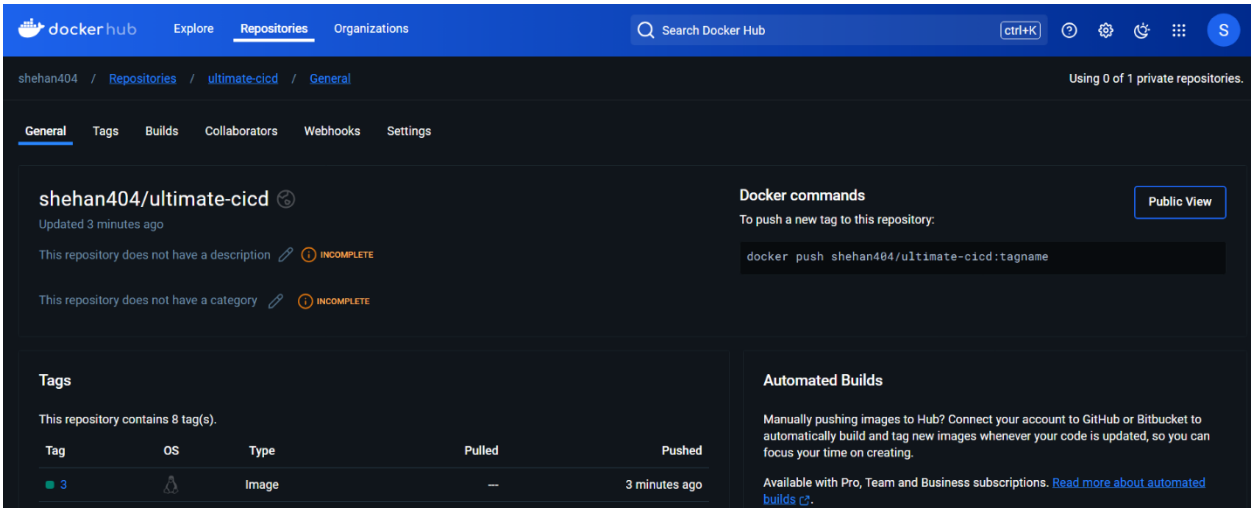
If an error occurs, refer the console output and try to troubleshoot the error. Then build again.

After all done well you will receive the following output.

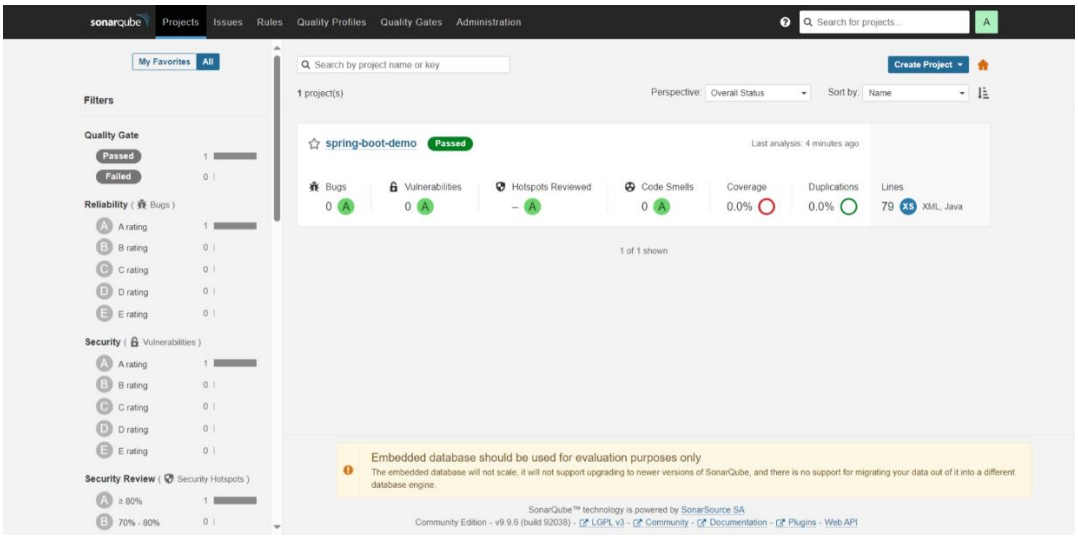
```
[Pipeline] }
[Pipeline] // withCredentials
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
$ docker stop --time=1 71f8b8cca3992d55a1318ec8a2f4db331b71443fefc32702e9cf58835a59eb3e
$ docker rm -f --volumes 71f8b8cca3992d55a1318ec8a2f4db331b71443fefc32702e9cf58835a59eb3e
[Pipeline] // withDockerContainer
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Now if you check your docker hub you can see a new docker image in a new repository.



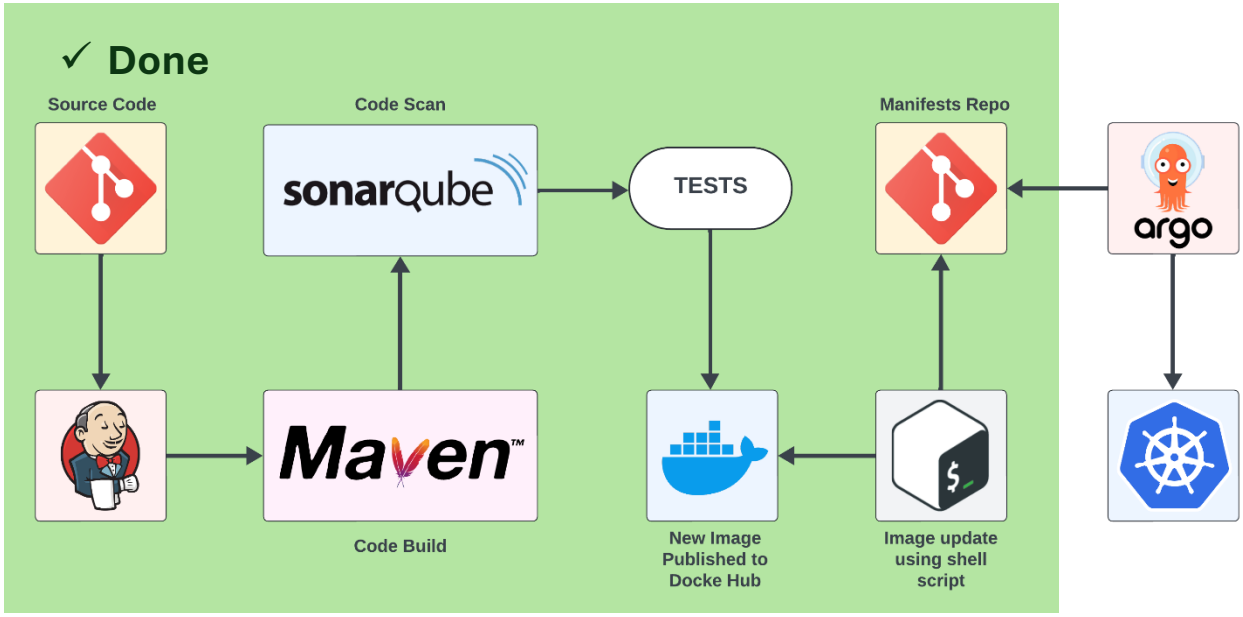


In SonarQube page under projects you can see like this



Which means no issues in the source code 😊.

Good Job. We have implemented CI part successfully 😊



Now Let's move on to CD part with ArgoCD.

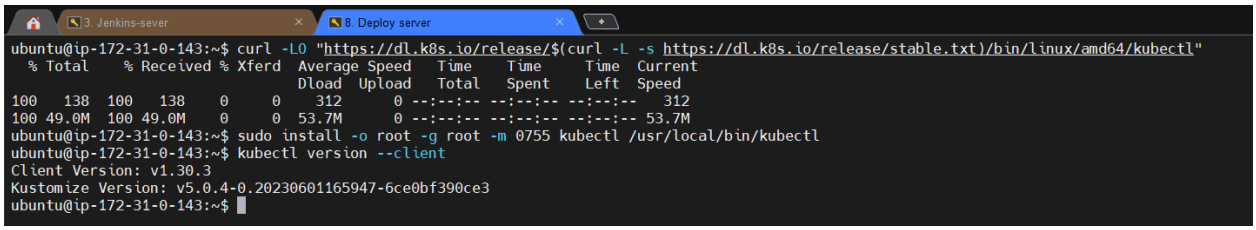
6. ArgoCD

Go to Deploy-server in MobaXterm

1. Install Kubectl

Run following commands

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
kubectl version –client
```



2. Install minicube

First install docker as we installed before. Because We use docker as driver for minicube.

Now run

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64
```

then run `minikube start --driver=docker`

```
ubuntu@ip-172-31-0-143:~$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 91.1M  100 91.1M    0     0  11.3M      0  0:00:07  0:00:07 --:--:-- 15.6M
ubuntu@ip-172-31-0-143:~$ minikube start --driver=docker
* minikube v1.33.1 on Ubuntu 24.04 (xen/amd64)
* 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 v0.0.44 ...
* Downloading Kubernetes v1.30.0 preload ...
    > preloaded-images-k8s-v18-v1...:  342.90 MiB / 342.90 MiB  100.00% 15.49 M
    > gcr.io/k8s-minikube/kicbase...:  481.58 MiB / 481.58 MiB  100.00% 15.09 M
* Creating docker container (CPUs=2, Memory=2200MB) ...
* Preparing Kubernetes v1.30.0 on Docker 26.1.1 ...
  - 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
ubuntu@ip-172-31-0-143:~$
```

Run `minikube status`

```
ubuntu@ip-172-31-0-143:~$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured

ubuntu@ip-172-31-0-143:~$
```

Cool minikube is running!

3. Setup Argocd

Run following command

```
kubectl create namespace argocd

kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

Now run `kubectl get pods -n argocd`

```
ubuntu@ip-172-31-0-143:~$ kubectl get pods -n argocd
NAME                                                READY   STATUS    RESTARTS   AGE
argocd-application-controller-0                     1/1     Running   0          49s
argocd-applicationset-controller-86c8767989-gtw2t   1/1     Running   0          49s
argocd-dex-server-55c6d584b5-q6s76                  1/1     Running   0          49s
argocd-notifications-controller-74965f8dc9-n7hwk    1/1     Running   0          49s
argocd-redis-5dcbd87d95-tdfm9                       1/1     Running   0          49s
argocd-repo-server-5b76df5967-5xgmn                 1/1     Running   0          49s
argocd-server-77f69c9d6-lj49d                       1/1     Running   0          49s
ubuntu@ip-172-31-0-143:~$
```

Cool pods are running.

Now run `kubectl get svc -n argocd`

```
ubuntu@ip-172-31-0-143:~$ kubectl get svc -n argocd
NAME                                      TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
argocd-applicationset-controller          ClusterIP   10.96.58.58      <none>        7000/TCP,8080/TCP            2m30s
argocd-dex-server                         ClusterIP   10.100.109.8     <none>        5556/TCP,5557/TCP,5558/TCP   2m30s
argocd-metrics                            ClusterIP   10.101.244.255   <none>        8082/TCP                     2m30s
argocd-notifications-controller-metrics   ClusterIP   10.104.4.131     <none>        9001/TCP                     2m30s
argocd-redis                              ClusterIP   10.104.14.117    <none>        6379/TCP                     2m30s
argocd-repo-server                        ClusterIP   10.108.7.14      <none>        8081/TCP,8084/TCP            2m30s
argocd-server                             ClusterIP   10.99.161.17     <none>        80/TCP,443/TCP               2m30s
argocd-server-metrics                     ClusterIP   10.107.237.252   <none>        8083/TCP                     2m30s
ubuntu@ip-172-31-0-143:~$
```

You can see that argocd-server is running at port 443

Let’s forward it to port 9090

Run `kubectl port-forward svc/argocd-server -n argocd 9090:443 --address 0.0.0.0 &`

```
ubuntu@ip-172-31-0-143:~$ kubectl port-forward svc/argocd-server -n argocd 9090:443 --address 0.0.0.0 &
[1] 18563
ubuntu@ip-172-31-0-143:~$ Forwarding from 0.0.0.0:9090 -> 8080
```

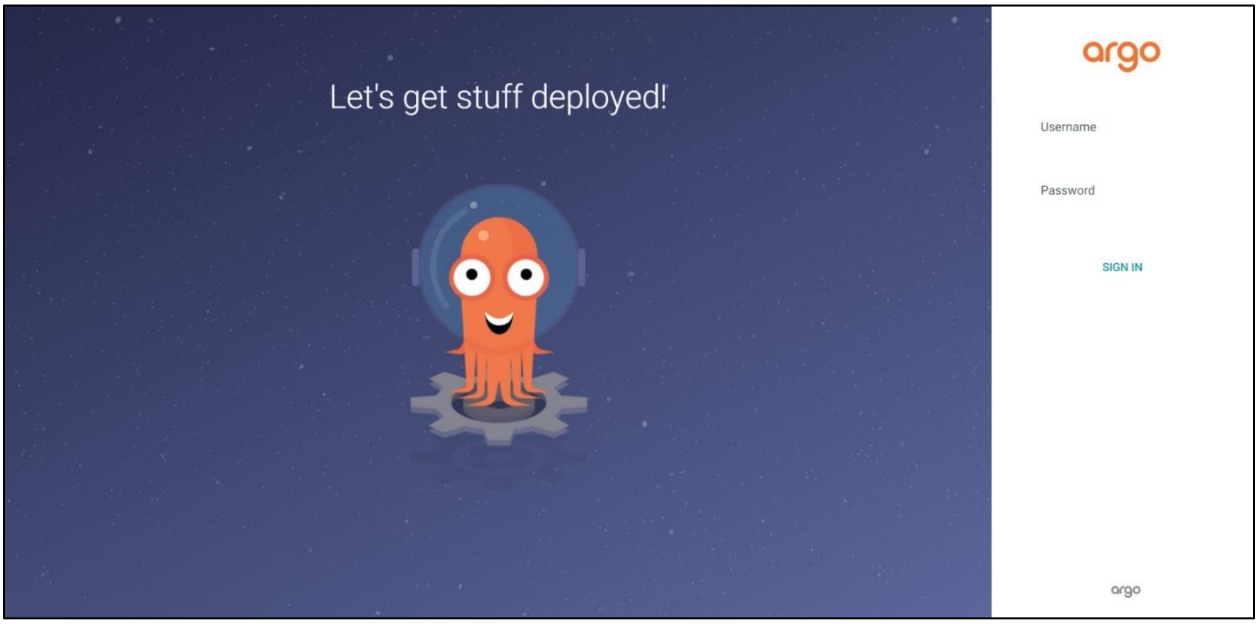
Now open the browser and enter following URL

`http://<Deploy-server ec2 instance public ip address>:9090`

In my case `https://3.111.52.39:9090/`

It will show that the page is not secure. Click advanced > Continue to 3.111.52.39 (unsafe)

Cool you entered to the ArgoCD Login page



Default username is admin.

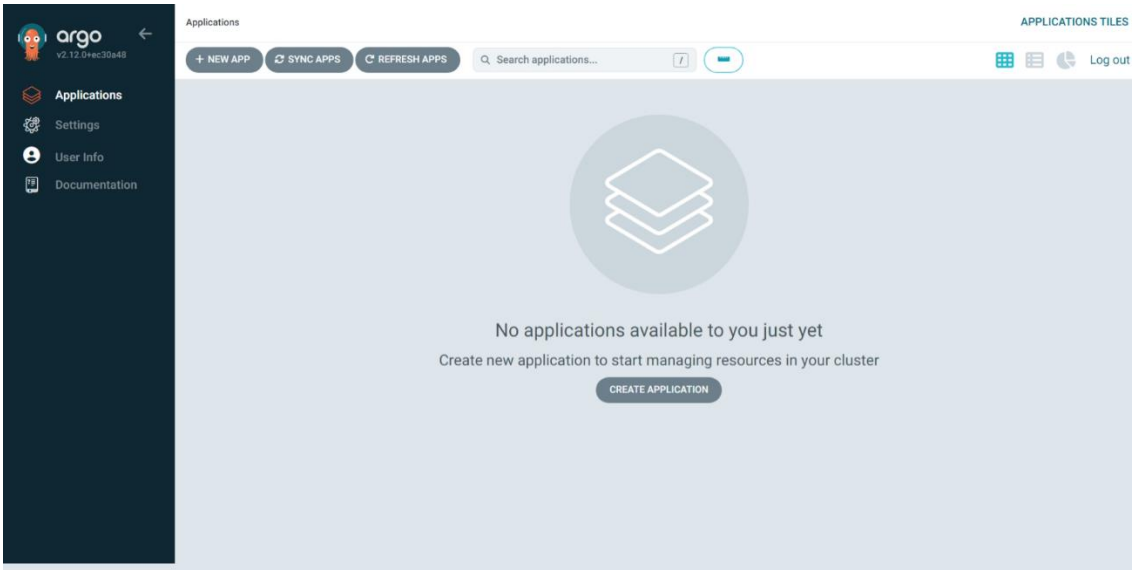
To obtain default password run following command

```
kubectl -n argocd get secret argocd-initial-admin-secret -o
jsonpath="{.data.password}" | base64 -d
```

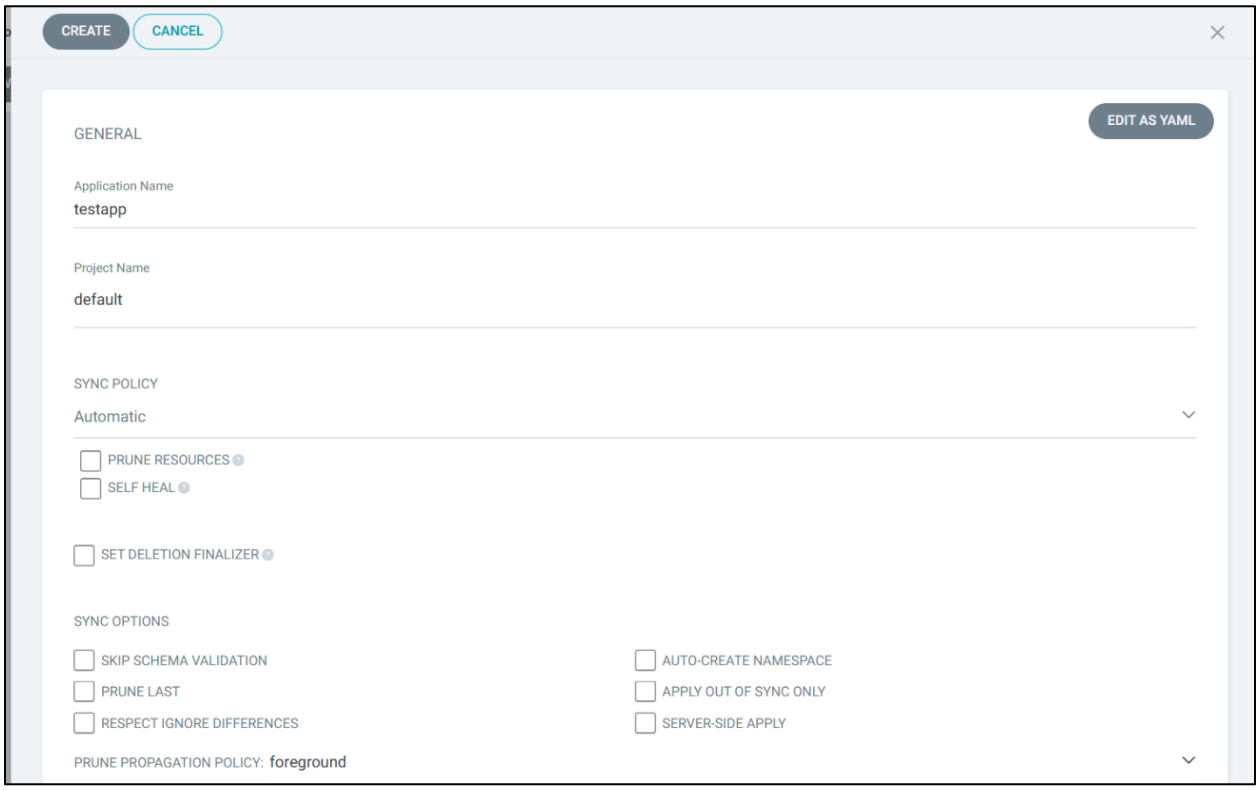
```
ubuntu@ip-172-31-0-143:~$ kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d
oX-UE85gkp4n6YYQubuntu@ip-172-31-0-143:~$
```

and here is the password: oX-UE85gkp4n6YYQ

Login to ArgoCD using them

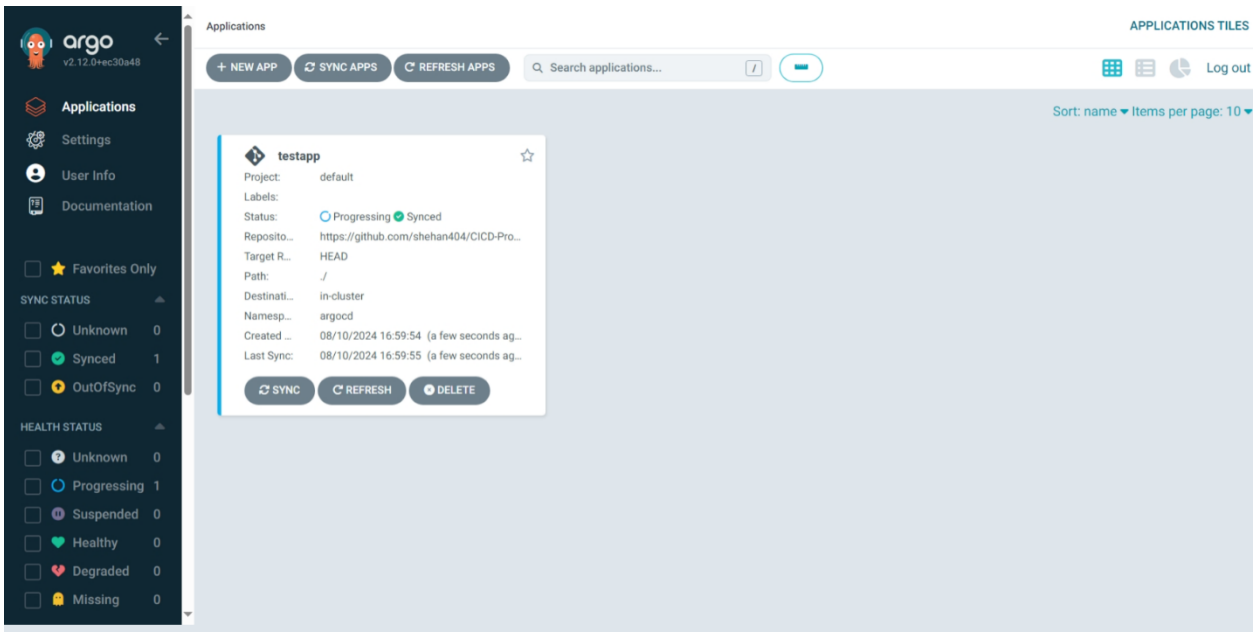


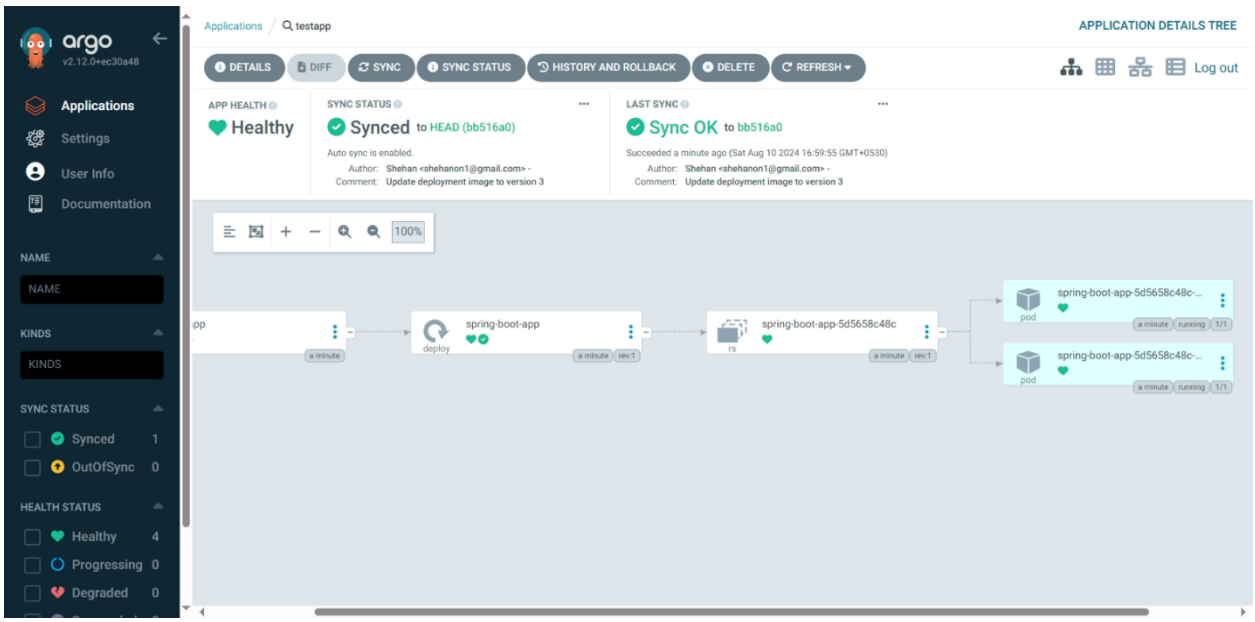
Go to NEW APP and insert following



Click create

You will see following page





You can see that everything is Synced and Healthy.

CD part is completed.

Congratulation!!! You have successfully implemented End to End CICD Pipeline
😊😊😊

Now let’s try to access the application using browser

Run `kubectl get pods -n argocd`

```
ubuntu@ip-172-31-0-143:~$ kubectl get pods -n argocd
NAME                                               READY   STATUS    RESTARTS   AGE
argocd-application-controller-0                    1/1     Running   0          28m
argocd-applicationset-controller-86c8767989-gtw2t  1/1     Running   0          28m
argocd-dex-server-55c6d584b5-q6s76                 1/1     Running   0          28m
argocd-notifications-controller-74965f8dc9-n7hwk   1/1     Running   0          28m
argocd-redis-5dcbd87d95-tdfm9                      1/1     Running   0          28m
argocd-repo-server-5b76df5967-5xgmn                1/1     Running   0          28m
argocd-server-77f69c9d6-lj49d                      1/1     Running   0          28m
spring-boot-app-5d5658c48c-d62qr                   1/1     Running   0          7m45s
spring-boot-app-5d5658c48c-vw74g                   1/1     Running   0          7m45s
```

Copy name of one pod

And run following command

```
kubectl expose pod spring-boot-app-5d5658c48c-d62qr --type=NodePort --port=8080 -n argocd
```

```
ubuntu@ip-172-31-0-143:~$  kubectl expose pod spring-boot-app-5d5658c48c-d62qr --type=NodePort --port=8080 -n argocd
service/spring-boot-app-5d5658c48c-d62qr exposed
```

Run `kubectl get svc -n argocd`

```
ubuntu@ip-172-31-0-143:~$ kubectl get svc -n argocd
NAME                                      TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
argocd-applicationset-controller          ClusterIP   10.96.58.58      <none>        7000/TCP,8080/TCP            31m
argocd-dex-server                         ClusterIP   10.100.109.8     <none>        5556/TCP,5557/TCP,5558/TCP   31m
argocd-metrics                            ClusterIP   10.101.244.255   <none>        8082/TCP                     31m
argocd-notifications-controller-metrics   ClusterIP   10.104.4.131     <none>        9001/TCP                     31m
argocd-redis                              ClusterIP   10.104.14.117    <none>        6379/TCP                     31m
argocd-repo-server                        ClusterIP   10.108.7.14      <none>        8081/TCP,8084/TCP            31m
argocd-server                             ClusterIP   10.99.161.17     <none>        80/TCP,443/TCP               31m
argocd-server-metrics                     ClusterIP   10.107.237.252   <none>        8083/TCP                     31m
spring-boot-app-5d5658c48c-d62qr          NodePort    10.98.225.195    <none>        8080:30177/TCP               3s
```

Now spring-boot-app-5d5658c48c-d62qr run on port 8080

Run following command for port forwarding

```
kubectl port-forward svc/spring-boot-app-5d5658c48c-d62qr -n argocd 8080:8080 --address 0.0.0.0 &
```

```
ubuntu@ip-172-31-0-143:~$  kubectl port-forward svc/spring-boot-app-5d5658c48c-d62qr -n argocd 8080:8080 --address 0.0.0.0 &
[2] 33661
ubuntu@ip-172-31-0-143:~$ Forwarding from 0.0.0.0:8080 -> 8080

ubuntu@ip-172-31-0-143:~$ curl 3.111.52.39:8080
```

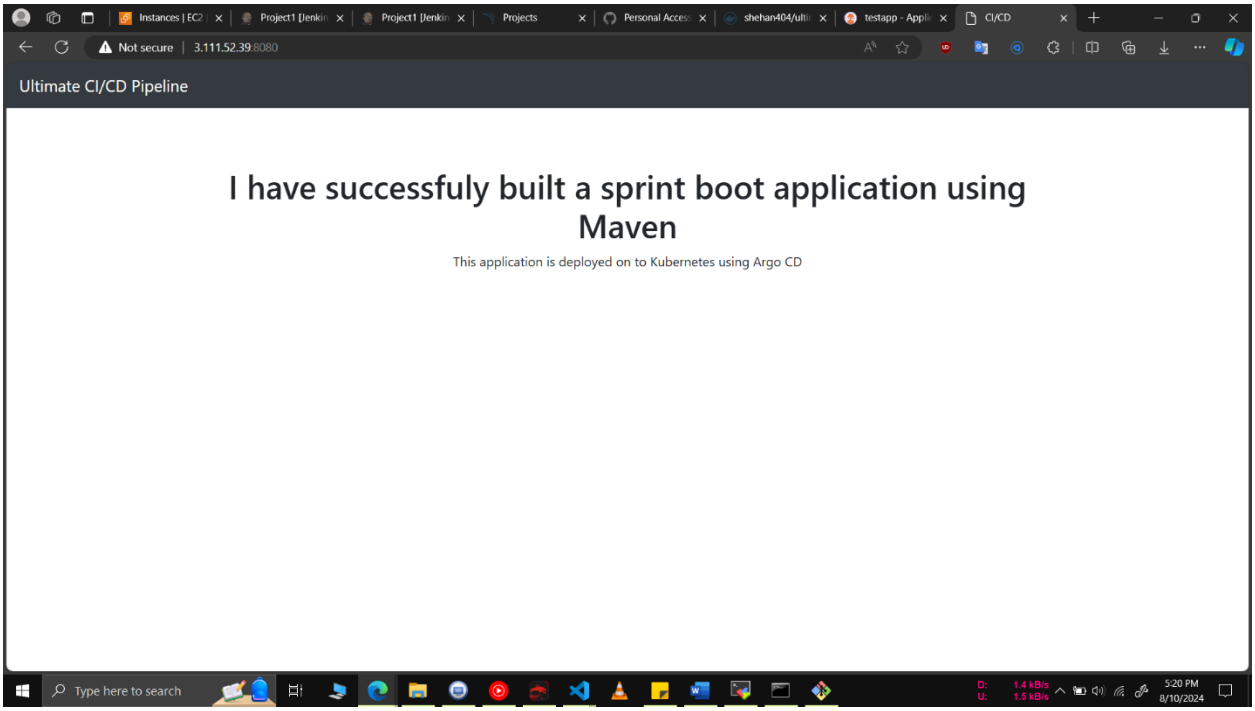
Now run `curl <deploy-server public ip address>:8080`

In my case `curl 3.111.52.39:8080`

If it gives following result, we can access it using browser

```
ubuntu@ip-172-31-0-143:~$ curl 3.111.52.39:8080
Handling connection for 8080
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"/>
    <link href="/css/main.css?" rel="stylesheet">
    <title>CI/CD</title>
</head>
<body>

<nav class="navbar navbar-expand-md navbar-dark bg-dark fixed-top">
    <a class="navbar-brand" href="#">Ultimate CI/CD Pipeline</div></a>
</nav>

<main role="main" class="container">
    <div class="starter-template">
        <h1>I have successfuly built a sprint boot application using Maven</h1>
        <p>This application is deployed on to Kubernetes using Argo CD</p>
    </div>
</main>

<script src="/js/main.js"></script>
</body>

</html>
```

Go to browser and paste URL used in curl command



Here is our application 😊 😊 😊

Note :

After completing the demonstration, shut down and terminate the EC2 instance; otherwise, you will incur additional costs.

KEEP LEARNING 😊 😊 😊