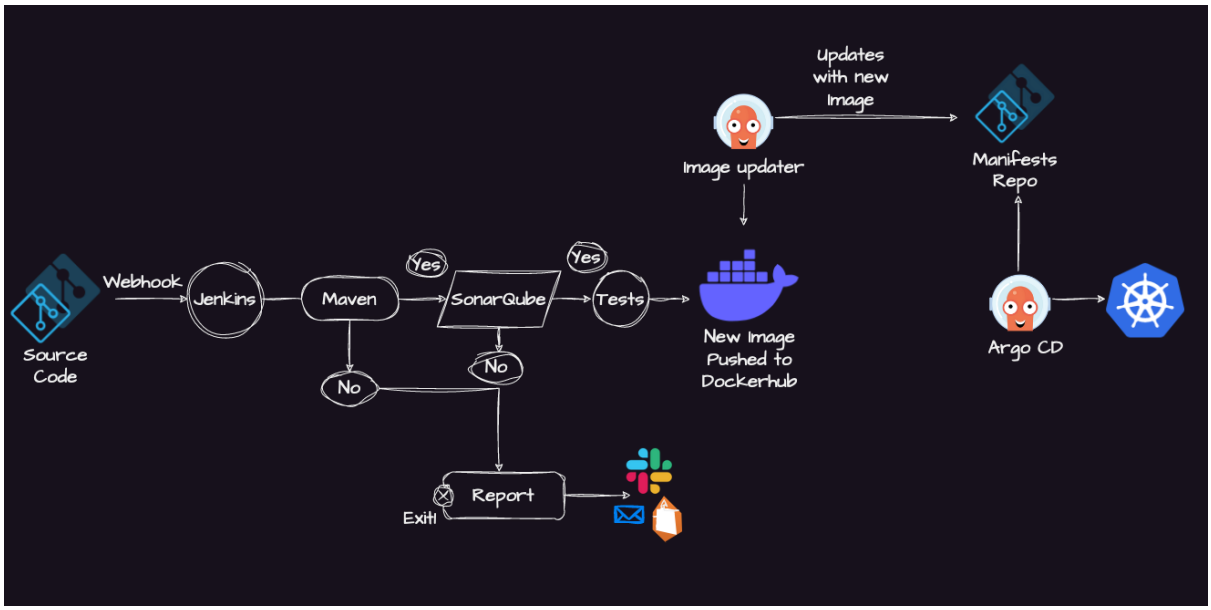
Project Documentation: CI/CD Pipeline Using Jenkins and ArgoCD

Project Overview

This project aims to set up a **CI/CD pipeline** using **Jenkins** for Continuous Integration (CI) and **ArgoCD** for Continuous Deployment (CD) in a Kubernetes environment. The pipeline automates the process of building, testing, and deploying applications, enabling faster, more reliable software delivery using modern DevOps practices.

Architecture



Tools and Technologies

- **Jenkins**: For automating the CI process (build, test, and package).
- **ArgoCD**: For continuous deployment (CD) and ensuring Kubernetes clusters match the desired state in Git.
- **Git**: For version control of both source code and Kubernetes manifests (e.g., GitHub).
- **Kubernetes**: For managing containerized applications.
- **Docker**: For containerization of applications.
- **Container Registry**: (e.g., DockerHub) to store Docker images

Prerequisites:

- Java application code hosted on a Git repository
- Jenkins server
- Kubernetes cluster
- Argo CD

Steps:

1. Install the necessary Jenkins plugins:

1.1 Git plugin

1.2 Maven Integration plugin

1.3 Pipeline plugin

1.4 Kubernetes Continuous Deploy plugin

2. Create a new Jenkins pipeline:

2.1 In Jenkins, create a new pipeline job and configure it with the Git repository URL for the Java application.

2.2 Add a Jenkins file to the Git repository to define the pipeline stages.

3. Define the pipeline stages:

Stage 1: Checkout: Placeholder, fetches code from the repository.

Stage 2: Build and Test: Builds the project and packages it.

Stage 3: Static Code Analysis: Runs static code analysis via SonarQube.

Stage 4: Build and Push Docker Image: Builds a Docker image and pushes it to a registry.

Stage 5: Update Deployment File: Updates the Kubernetes deployment manifest with the new Docker image version and pushes it back to Git.

4. Configure Jenkins pipeline stages:

Stage 1: Use the Git plugin to check out the source code from the Git repository.

Stage 2: Use the Maven Integration plugin to build the Java application.

Stage 3: Use the SonarQube plugin to analyze the code quality of the Java application.

Stage 4. Build a Docker image for the application and push it to a Docker registry.

Stage 5: Update the Kubernetes deployment manifest file to use the new Docker image version

5. Set up Argo CD:

Install Argo CD on the Kubernetes cluster.

Set up a Git repository for Argo CD to track the changes in the Kubernetes manifests.

6. Configure Jenkins pipeline to integrate with Argo CD:

6.1 Add the Argo CD API token to Jenkins credentials.

6.2 Update the Jenkins pipeline to include the Argo CD deployment stage.

7. Run the Jenkins pipeline:

7.1 Trigger the Jenkins pipeline to start the CI/CD process for the Java application.

7.2 Monitor the pipeline stages and fix any issues that arise.

This end-to-end Jenkins pipeline will automate the entire CI/CD process for a Java application, from code checkout to production deployment, using popular tools like SonarQube, Argo CD, Helm, and Kubernetes.

Demo

Launch an EC2 Instance

- Go to AWS Console
- t2.2xlarge
- Launch instances

Install Jenkins.

Pre-Requisites:

- Java (JDK)

Run the below commands to install Java and Jenkins

Install Java

sudo apt update

sudo apt install openjdk-17-jre

java –version

```
openjdk 17.0.12 2024-07-16
```

Now, you can proceed with installing Jenkins

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

jenkins –version

```
2.480
```

**Note: ** By default, Jenkins will not be accessible to the external world due to the inbound traffic restriction by AWS. Open port 8080 in the inbound traffic rules as show below.

- EC2 > Instances > Click on
- In the bottom tabs -> Click on Security
- Security groups

Add inbound traffic rules as shown in the image (you can just allow TCP 8080 as well, in my case, I allowed (ALL TRAFFIC)

Login to Jenkins using the below URL:

http://:8080 [You can get the ec2-instance-public-ip-address from your AWS EC2 console page]

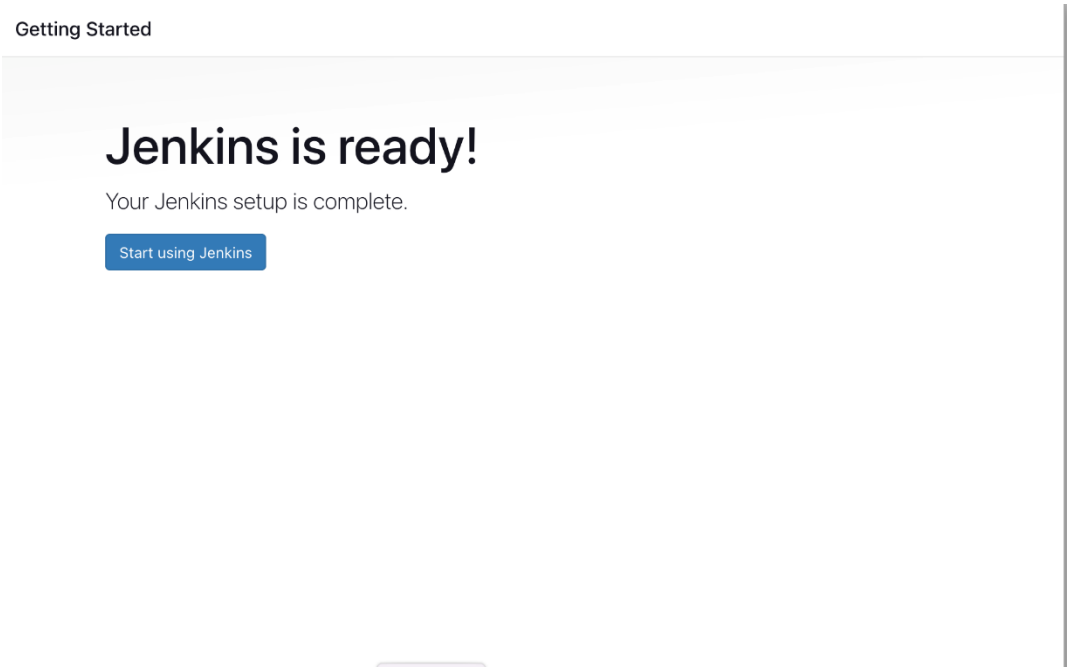
http://98.82.15.5:8080

After you login to Jenkins, - Run the command to copy the Jenkins Admin Password –

`sudo cat /var/lib/jenkins/secrets/initialAdminPassword` - Enter the Administrator password

Click on Install suggested plugins

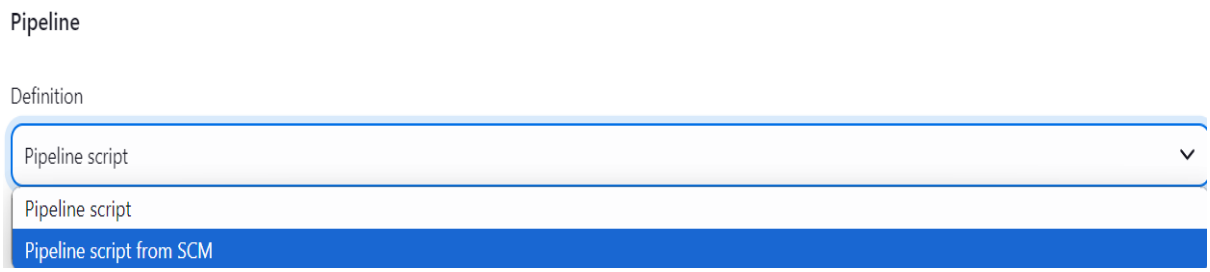
Jenkins Installation is Successful. You can now starting using the Jenkins



Create a Jenkins pipeline that retrieves a pipeline script from SCM (like Git) and executes it, follow these steps:

1. Set Up a Jenkins Pipeline Job:

1. Log in to Jenkins and click on "New Item".
2. Enter a name for your job, select "Pipeline", and click OK.
3. **Configure the Pipeline Job:**



2. In the Pipeline section:

    - Set **Definition** to **Pipeline script from SCM**.
    - Choose your SCM (e.g., Git).
    - Enter your repository URL.
    - Configure the **branch** to track. (main branch)
    - If needed, enter **credentials** for your SCM (for private repositories).
    - Specify the location of the pipeline script (usually a Jenkinsfile).

Example Configuration:

- **SCM**: Git
- **Repository URL**:

  https://github.com/GopalKdevops/Jenkins-Zero-To-Hero

- **Branch**: */main
- **Script Path**: Jenkinsfile

  java-maven-sonar-argocd-helm-k8s/spring-boot-app/JenkinsFile

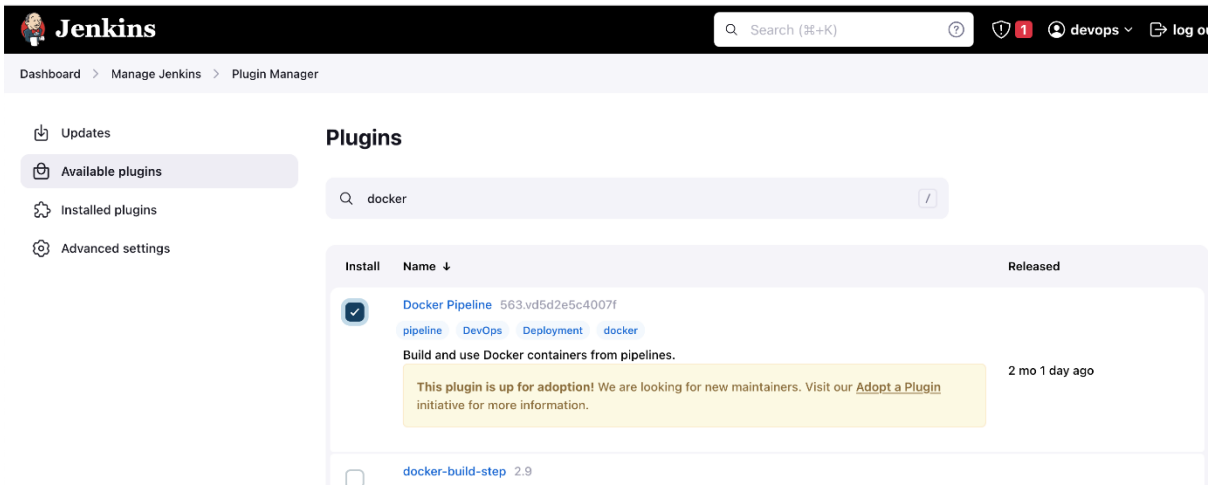
3. Create the Jenkinsfile in Your Repository:

This is where you define the pipeline stages and steps.

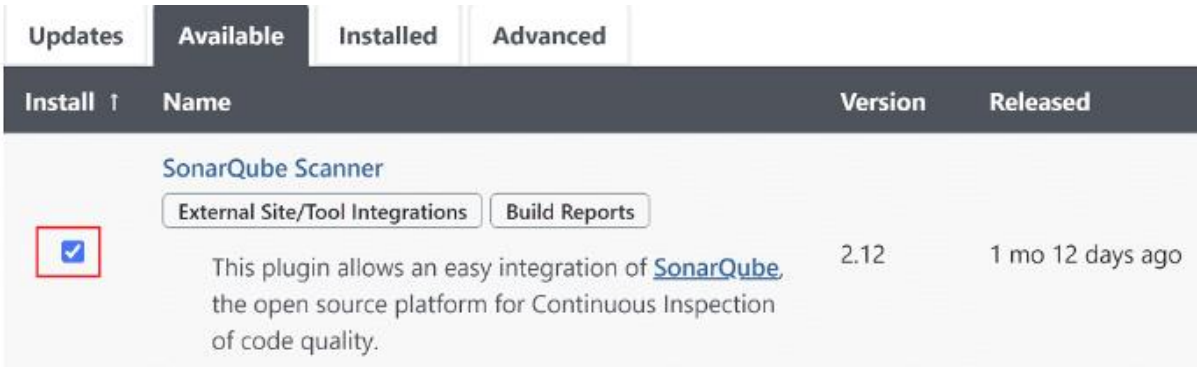
One of the good ways of writing the Jenkins pipeline is to use docker agent, In this demo we use docker container as agent of our Jenkins pipeline. So we need to install docker plugins in our pipeline

Install the Docker Pipeline plugin in Jenkins:

- Log in to Jenkins.
- Go to Manage Jenkins > Manage Plugins.
- In the Available tab, search for "Docker Pipeline".
- Select the plugin and click the Install button.
- Restart Jenkins after the plugin is installed.



Install SonarQube Scanner plugin



Configure a Sonar Server locally

sudo apt install unzip

adduser sonarqube

sudo su - 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

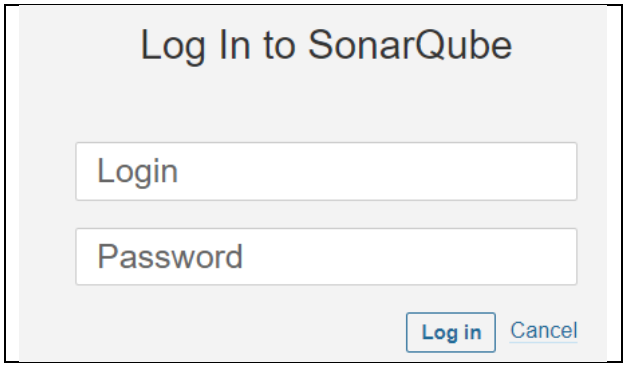
cd sonarqube-9.4.0.54424/bin/linux-x86-64/

./sonar.sh start

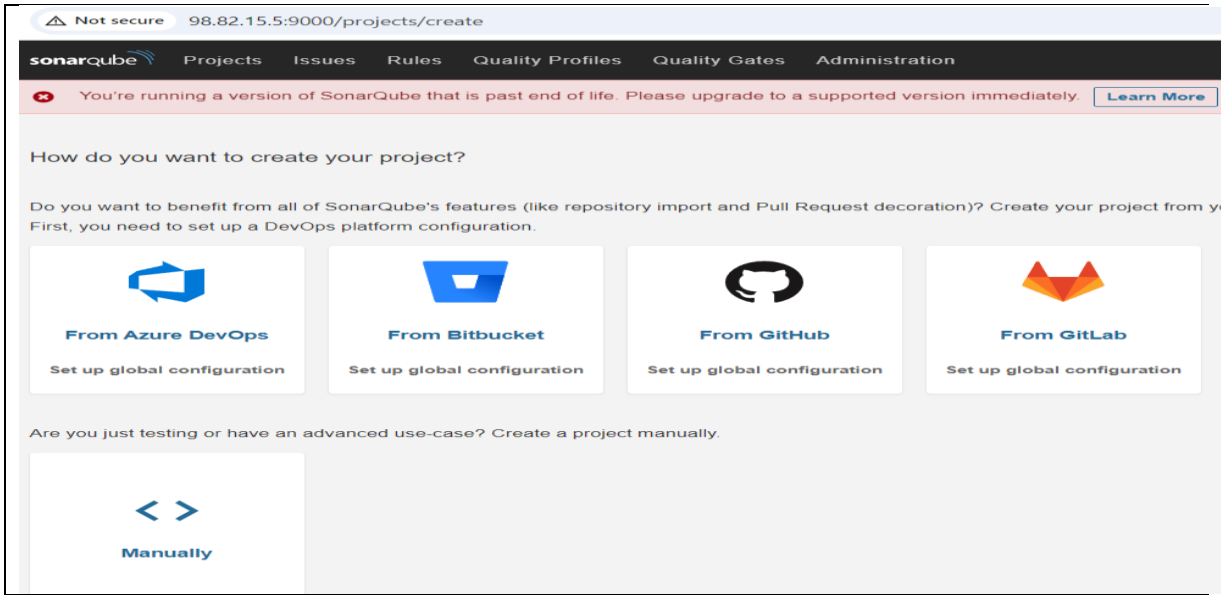
Login to Sonarqube using the below URL:

http://:9000 [You can get the ec2-instance-public-ip-address from your AWS EC2 console page]

http://98.82.15.5:9000/



Login and Password: admin



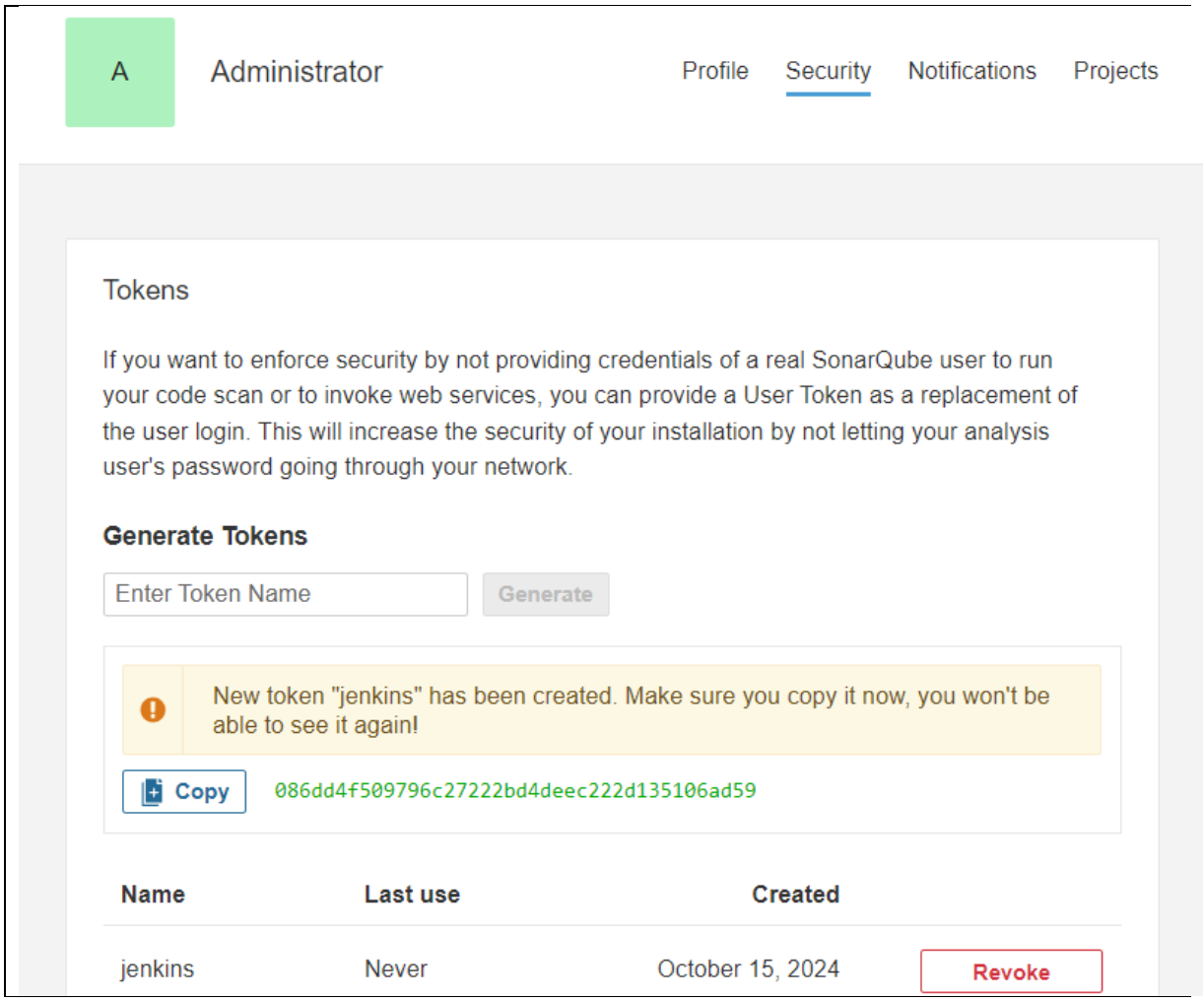
How Jenkins authenticated with sonarqube, because there are 2 different applications

Steps to Authenticate Sonarqube with Jenkins

Go to Sonarqube console

Click MyAccount -→ Select Security option →Click the Generate Token option and enter your name as your own, I chose Jenkins

Jenkins



Go to Jenkins console

Click on Manage jenkins →Select Manage Credentials → Go to system→Click Global Credentials → Add credentials → New credentials

Kind →Secret text

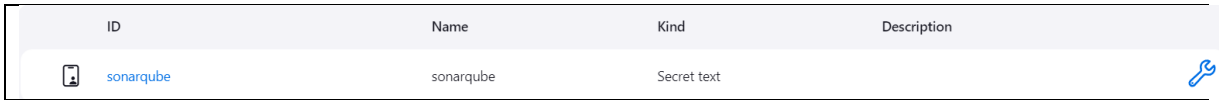
Secret

****

Id

Sonarqube

Click the Create button



Sonarqube configuration was successfully done.

Install Docker

On the EC2 instance, we install Docker

sudo apt install docker.io

docker –version

Grant Jenkins user and Ubuntu user permission to docker deamon.

sudo su -

usermod -aG docker jenkins

usermod -aG docker ubuntu

systemctl restart docker

Once all are done with the above steps, it is better to restart Jenkins.

http://<ec2-instance-public-ip>:8080/restart

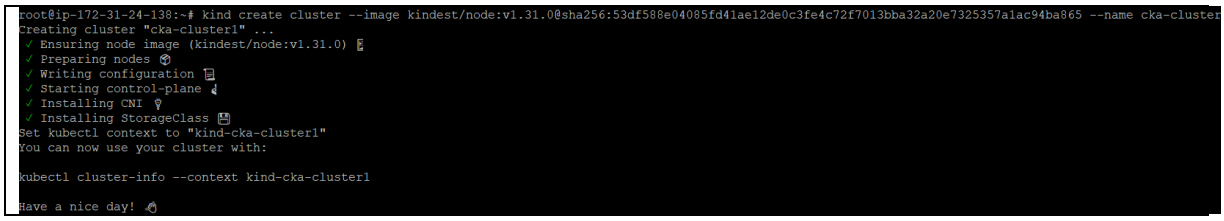
Install Kubernetes cluster with Kind

1.Install kind

curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-amd64
chmod +x ./kind
sudo mv ./kind /usr/local/bin/kind
kind –version

2.Create Kubernetes Cluster

kind create cluster --image kindest/node:v1.31.0@sha256:53df588e04085fd41ae12de0c3fe4c72f7013bba32a20e7325357a1ac94ba865 --name cka-cluster1



Install kubectl

curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

curl -LO https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl.sha256

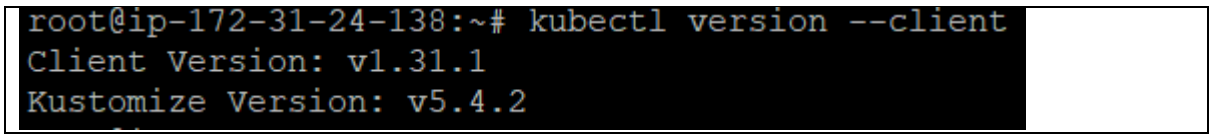
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl

chmod +x kubectl

mkdir -p ~/.local/bin

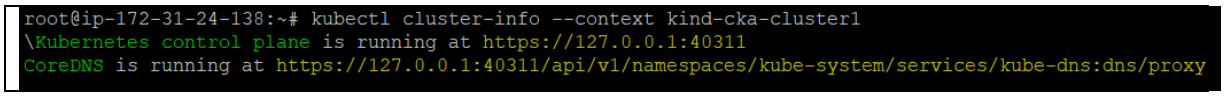
mv ./kubectl ~/.local/bin/kubectl

kubectl version –client



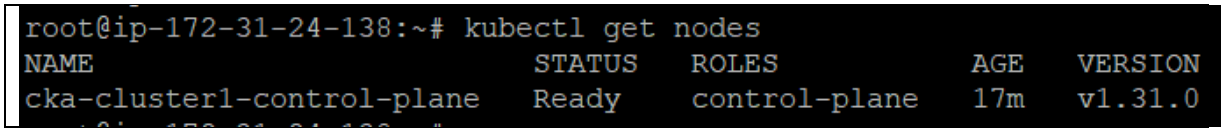
To Check the cluster is running

kubectl cluster-info --context kind-cka-cluster1



To check the nodes

Kubectl get nodes



Install ArgoCD

ArgoCD is an open-source, declarative GitOps continuous delivery tool for Kubernetes. When integrated with Jenkins in a CI/CD pipeline, **ArgoCD** serves a critical role in automating the deployment process, especially in a Kubernetes environment.

ArgoCD takes over the continuous deployment (CD) role, automating the deployment of the application to the Kubernetes cluster once the changes are committed to the Git repository.

Go to **https://operatorhub.io/**

Install on Kubernetes

Install Operator Lifecycle Manager (OLM), a tool to help manage the Operators running on your cluster.

curl -sL https://github.com/operator-framework/operator-lifecycle-manager/releases/download/v0.28.0/install.sh | bash -s v0.28.0

```
clusterserviceversion.operators.coreos.com/packageserver created
catalogsource.operators.coreos.com/operatorhubio-catalog created
Waiting for deployment "olm-operator" rollout to finish: 0 of 1 updated replicas are available...
deployment "olm-operator" successfully rolled out
Waiting for deployment "catalog-operator" rollout to finish: 0 of 1 updated replicas are available
deployment "catalog-operator" successfully rolled out
Package server phase: Installing
Package server phase: Succeeded
```

Install the operator by running the following command

kubectl create -f https://operatorhub.io/install/argocd-operator.yaml

This Operator will be installed in the "operators" namespace and will be usable from all namespaces in the cluster

```
subscription.operators.coreos.com/my-argocd-operator created
```

After install, watch your operator come up using next command.

kubectl get csv -n operators

```
gopal@DESKTOP-7B838JU:~$  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   Pending
gopal@DESKTOP-7B838JU:~$
gopal@DESKTOP-7B838JU:~$ 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
```

Initial Phase is Pending and then “Succeeded

Check the pods are created in the operator namespace

kubectl get pods –n operators

```
gopal@DESKTOP-7B838JU:~$ kubectl get pods -n operators
NAME                                                READY   STATUS    RESTARTS   AGE
argocd-operator-controller-manager-b74f795c6-d79bb   1/1     Running   0          7m30s
```

Review the Stages in the Jenkins file

1.Review the Docker Build and push stage

stage('Build and Push Docker Image') {

environment {

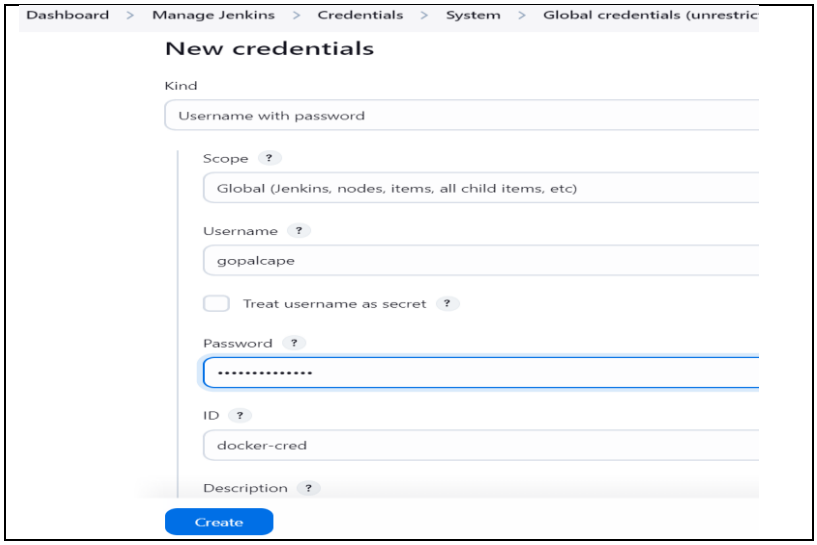
DOCKER_IMAGE = "abhishekf5/ultimate-cicd:${BUILD_NUMBER}"

// DOCKERFILE_LOCATION = "java-maven-sonar-argocd-helm-k8s/spring-boot-app/Dockerfile"

REGISTRY_CREDENTIALS = credentials('**docker-cred**')

We should update the above Docker registry credentilas (docker-cred)in the Jenkins pipeline

Manage Jenkins → Credentials → System → Global credentials → New credentials



2.Review the Deployment stage

stage('Update Deployment File') {

environment {

GIT_REPO_NAME = "Jenkins-Zero-To-Hero"

GIT_USER_NAME = "iam-veeramalla"

}

```
steps {
    withCredentials([string(credentialsId: 'github', variable: 'GITHUB_TOKEN')])
```

We should update the above GitHub Token credentilas in the Jenkins pipeline

Kind: Secret text

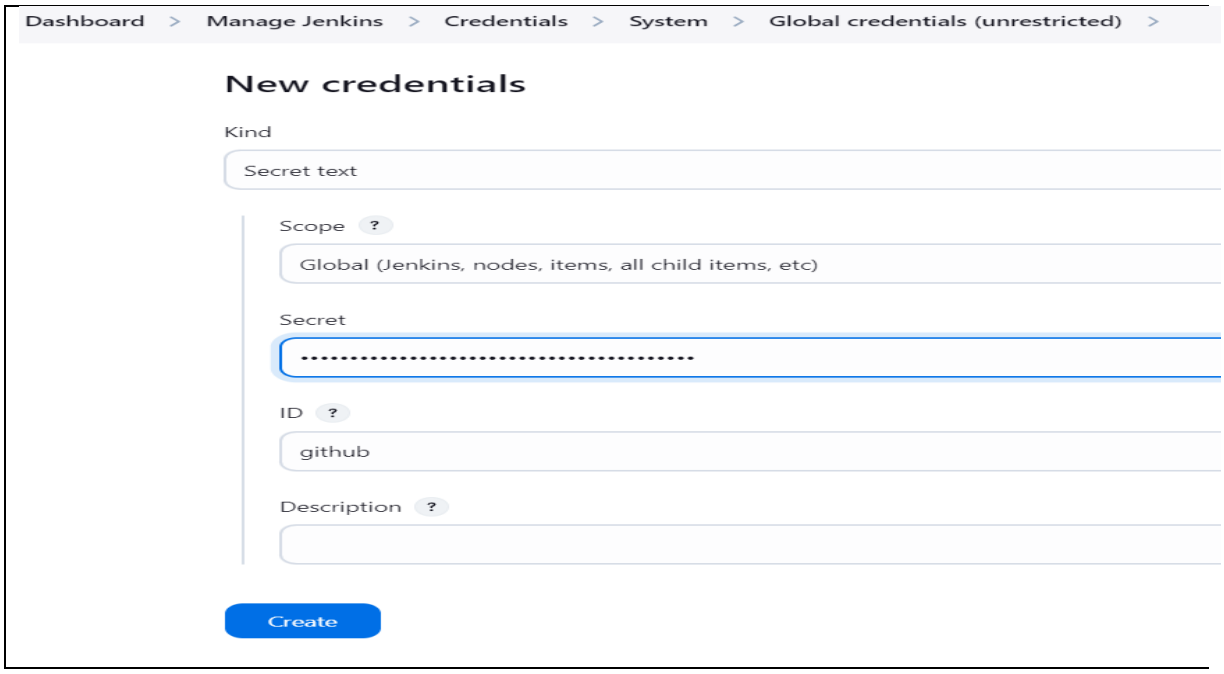
ID: github

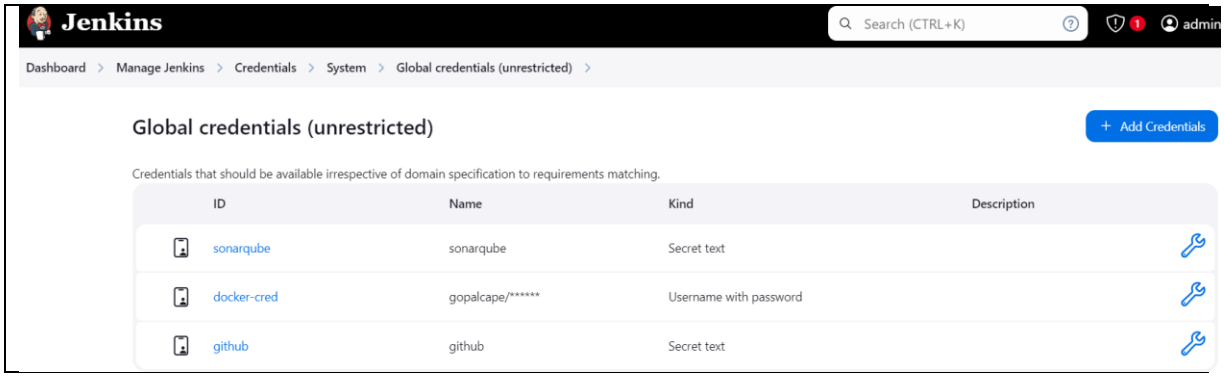
Secret :

For creating a Secret Token, Go to Github

Settings → developer settings → Personal Access Token → Token classic -→ Generate new token classic

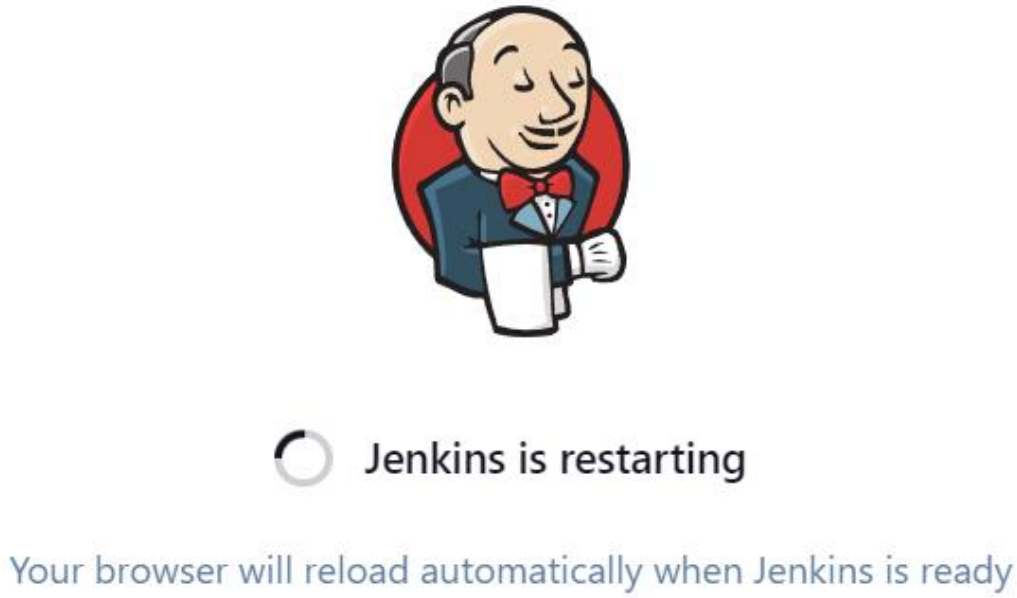
Update the token in the secret in the pipeline



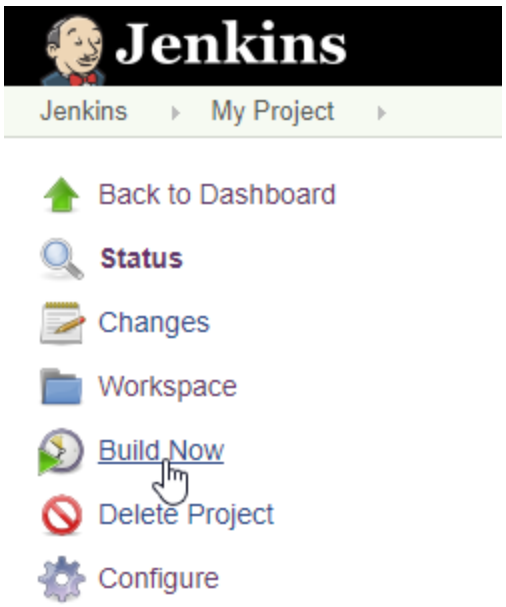


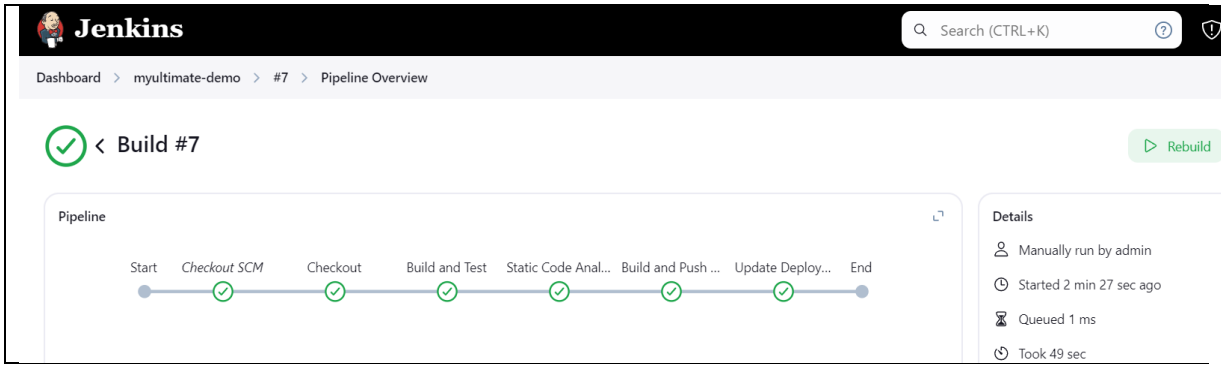
Once all are done with the above steps, it is better to restart Jenkins.

http://<ec2-instance-public-ip>:8080/restart

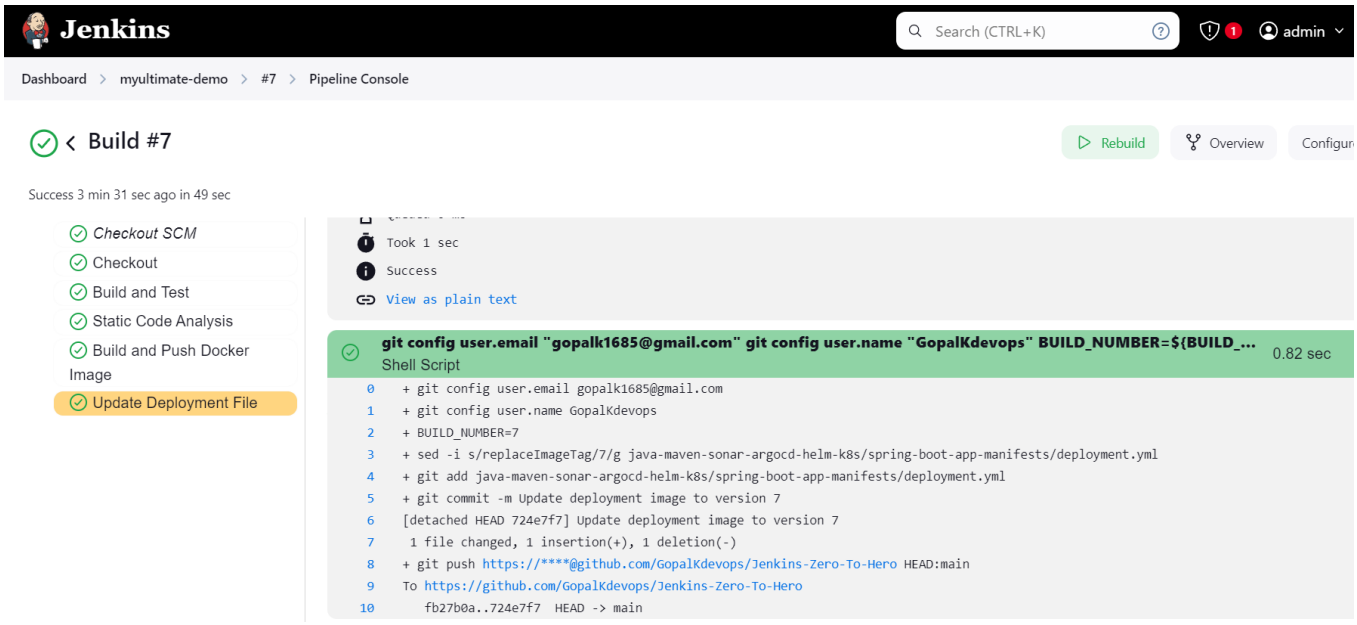


Click the Build now button to check the pipeline is working properly

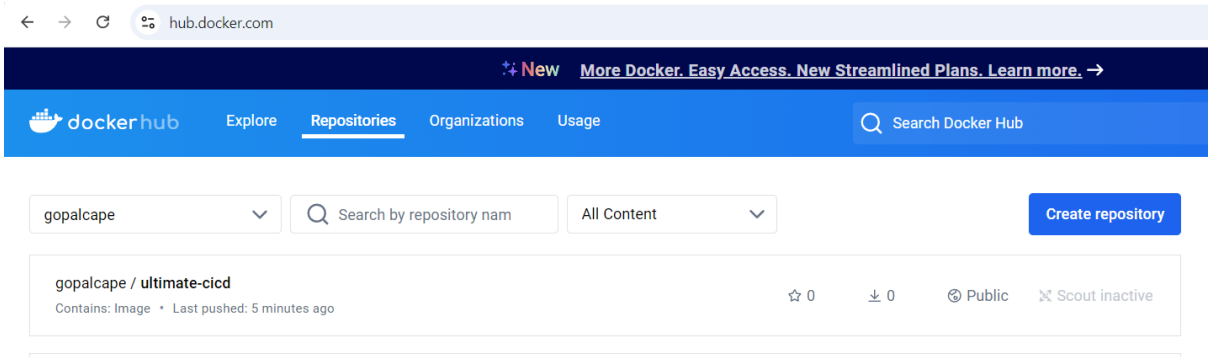




Pipeline run successfully



Check the latest images in DockerHub and the EC2 Instance



]docker images

```
root@ip-172-31-24-138:~# docker images
REPOSITORY                      TAG       IMAGE ID       CREATED          SIZE
gopalcape/ultimate-cicd         7         96513db64939   6 minutes ago    170MB
```

We have finished the stages of Jenkins continuous integration.

1.Checkout is done

2.Maven it has created the JAR file and download all the dependies

3,Sonarqube part also completed

4.Docker image also created

5.Shell script updated in the repo (deployment.yaml)

Final stage is use ArgoCD to deploy automatically on to the Kubernetes cluster

Go to the below link

https://argocd-operator.readthedocs.io/en/latest/usage/basics/

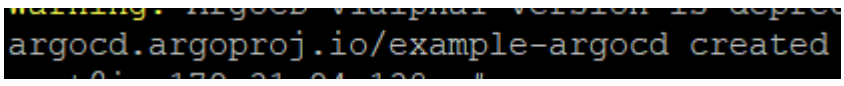
Create a new file argocd-basic.yml with the following content

It's a ArgoCD controller

```
apiVersion: argoproj.io/v1alpha1
kind: ArgoCD
metadata:
  name: example-argocd
  labels:
    example: basic
spec: {}
```

The purpose of this specific command is to create or update resources in the Kubernetes cluster that are defined in the argocd-basic.yml

kubectl apply –f argocd-basic.yml



To check the pods are created

kubectl get pods

```
root@ip-172-31-24-138:~# kubectl get pods
NAME                                          READY   STATUS    RESTARTS   AGE
example-argocd-application-controller-0       1/1     Running   0          72s
example-argocd-redis-6545fd6d6c-v55bv         1/1     Running   0          72s
example-argocd-repo-server-869d5757c7-5gtcd   1/1     Running   0          72s
example-argocd-server-76bb84cddc-m8mm9        1/1     Running   0          72s
```

Services in Kubernetes are responsible for enabling network access to a set of Pods, either within the cluster or from outside the cluster, depending on the service type.

kubectl get svc

```
root@ip-172-31-24-138:~# kubectl get svc
NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)             AGE
example-argocd-metrics        ClusterIP   10.96.53.60     <none>        8082/TCP            3m39s
example-argocd-redis          ClusterIP   10.96.182.232   <none>        6379/TCP            3m39s
example-argocd-repo-server    ClusterIP   10.96.149.105   <none>        8081/TCP,8084/TCP   3m39s
example-argocd-server         ClusterIP   10.96.57.196    <none>        80/TCP,443/TCP      3m39s
example-argocd-server-metrics ClusterIP   10.96.215.117   <none>        8083/TCP            3m39s
kubernetes                    ClusterIP   10.96.0.1       <none>        443/TCP             104m
```

Output: **example-argocd-server** is responsible for ArgoCD UI

kubectl edit svc example-argocd-server

Change the type into NodePort

```
service/example-argocd-server edited
```

kubectl get svc

```
root@ip-172-31-24-138:~# kubectl get svc
NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE
example-argocd-metrics        ClusterIP   10.96.53.60     <none>        8082/TCP                     8m34s
example-argocd-redis          ClusterIP   10.96.182.232   <none>        6379/TCP                     8m34s
example-argocd-repo-server    ClusterIP   10.96.149.105   <none>        8081/TCP,8084/TCP            8m34s
example-argocd-server         NodePort    10.96.57.196    <none>        80:30170/TCP,443:32703/TCP   8m34s
example-argocd-server-metrics ClusterIP   10.96.215.117   <none>        8083/TCP                     8m34s
kubernetes                    ClusterIP   10.96.0.1       <none>        443/TCP                      109m
```

Below command is used to access the ArgoCD server service running in a Kubernetes cluster.

To get the Nodeip run the below command

docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' cka-cluster1-control-plane

172.18.0.2

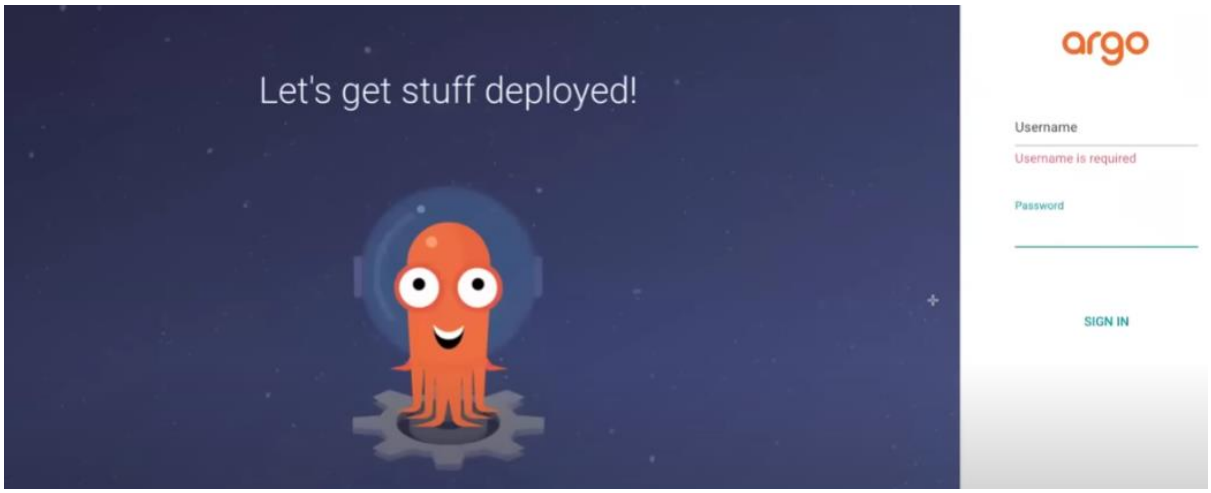
```
root@ip-172-31-24-138:~# docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' cka-cluster1-control-plane
172.18.0.2
```

To get the NodePort run the below command

kubectl get svc example-argocd-server

```
root@ip-172-31-24-138:~# kubectl get svc example-argocd-server
NAME                    TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)                      AGE
example-argocd-server   NodePort   10.96.57.196   <none>        80:30170/TCP,443:32703/TCP   96m
```

http://172.18.0.2:32703



How to get the ArgoCD password

kubectl get secret

```
root@ip-172-31-24-138:~# kubectl get secret
NAME                                  TYPE                 DATA   AGE
argocd-secret                         Opaque               5      103m
example-argocd-ca                     kubernetes.io/tls    3      103m
example-argocd-cluster                Opaque               1      103m
example-argocd-default-cluster-config Opaque               4      103m
example-argocd-redis-initial-password Opaque               2      103m
example-argocd-tls                    kubernetes.io/tls    2      103m
```

To view the secret put the below command

kubectl edit secret example-argocd-cluster

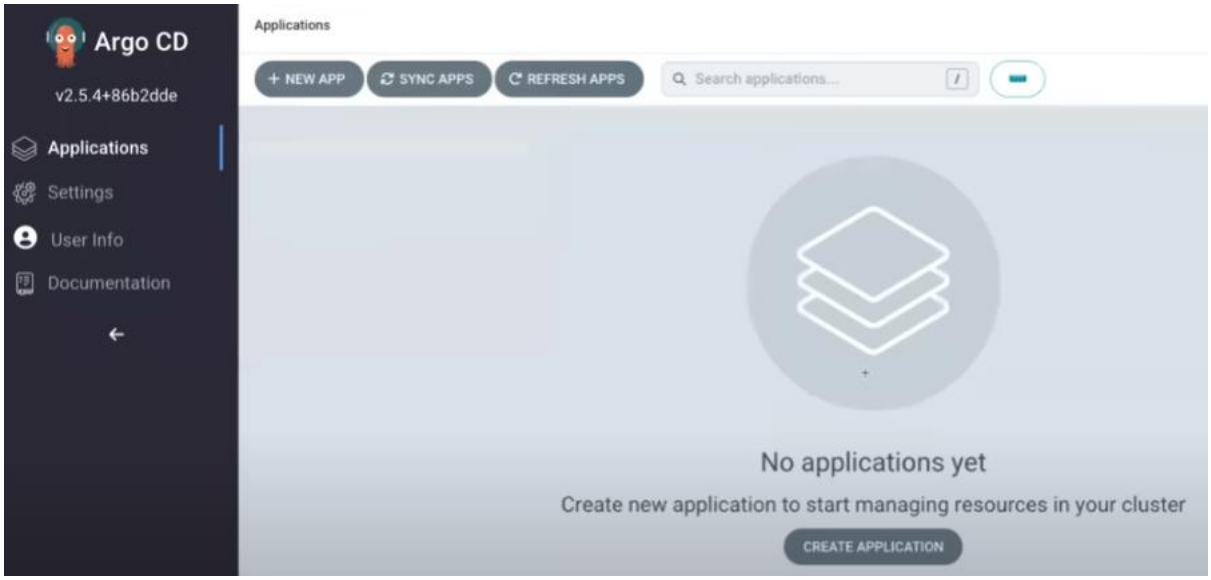
```
apiVersion: v1
data:
  admin.password: QmNBSEVTVTc2RHpSTGludzJhV092c2JnMHVvcE1QOFY=
kind: Secret
```

copy the secret QmNBSEVTVTc2RHpSTGludzJhV092c2JnMHVvcE1QOFY=

echo QmNBSEVTVTc2RHpSTGludzJhV092c2JnMHVvcE1QOFY= | base64 –d

we get the password

Enter the credentials ,now we successfully login into the ArgoCD UI



Fill all the appropriate details

1.Appname: Test

2.Project name: default

3.Sync: Automatic

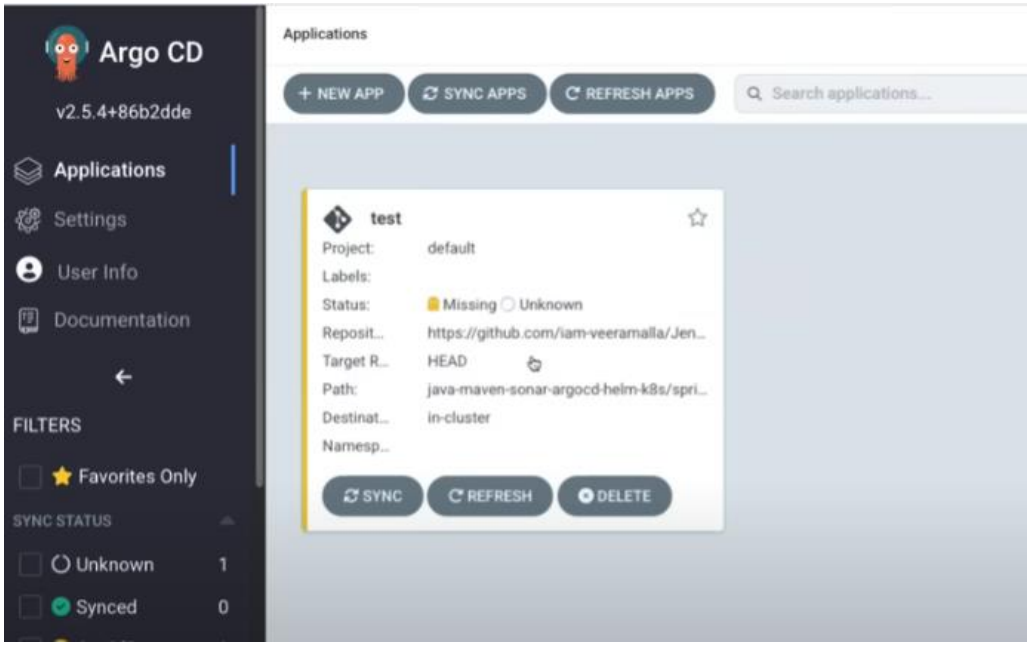
4.Repository url: github url

5.Path: Path of the deployment file

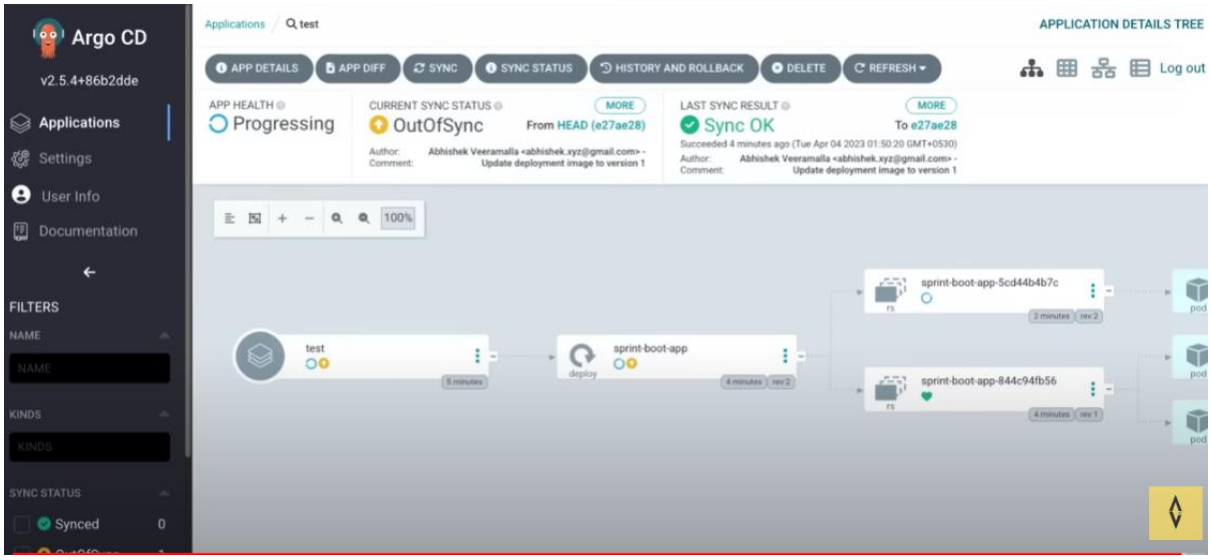
6.Cluster url: https://kubernetes-default-svc

7.namespace: default

Click Create Button



ArgoCD successfully deployed the spring boot applications



To check the deployments

kubectl get deploy

```
root@ip-172-31-24-138:~# kubectl get deploy
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
example-argocd-redis          1/1     1            1           127m
example-argocd-repo-server    1/1     1            1           127m
example-argocd-server         1/1     1            1           127m
```

Conclusion of the Project Using Jenkins as CI and ArgoCD as CD

The integration of **Jenkins** for Continuous Integration (CI) and **ArgoCD** for Continuous Deployment (CD) results in a highly efficient and automated CI/CD pipeline. Jenkins handles the build, testing, and verification stages, ensuring that only reliable and validated code is packaged and prepared for deployment. ArgoCD then takes over by automatically deploying these changes to Kubernetes environments using GitOps principles.

The key outcomes of this integration include:

- **End-to-End Automation**: The entire software delivery process, from code commit to deployment, is fully automated, minimizing manual intervention and reducing human error.
- **Increased Reliability**: With Jenkins validating the code and ArgoCD ensuring consistency between Git and Kubernetes clusters, the system is more stable and secure.

- **Faster Delivery**: Continuous integration and deployment speed up the delivery of new features, updates, and bug fixes, enabling more frequent releases and faster feedback loops.
- **GitOps-Based Deployment**: ArgoCD's declarative, GitOps-based approach ensures that the desired state of Kubernetes clusters is always tracked and recoverable, making it easier to roll back changes or restore consistency in case of issues.
- **Scalability and Flexibility**: The combination of Jenkins and ArgoCD provides a scalable solution for managing multiple applications and environments, making it suitable for both small and large-scale projects.

By leveraging Jenkins and ArgoCD together, the project achieves a modern, streamlined CI/CD workflow that improves efficiency, accelerates delivery, and enhances the overall quality of software deployments.