**Deploying Spring Boot Application on Kubernetes using docker and jenkins**

**Git url: https://github.com/harishdasari1348/springboot-microservices.git**

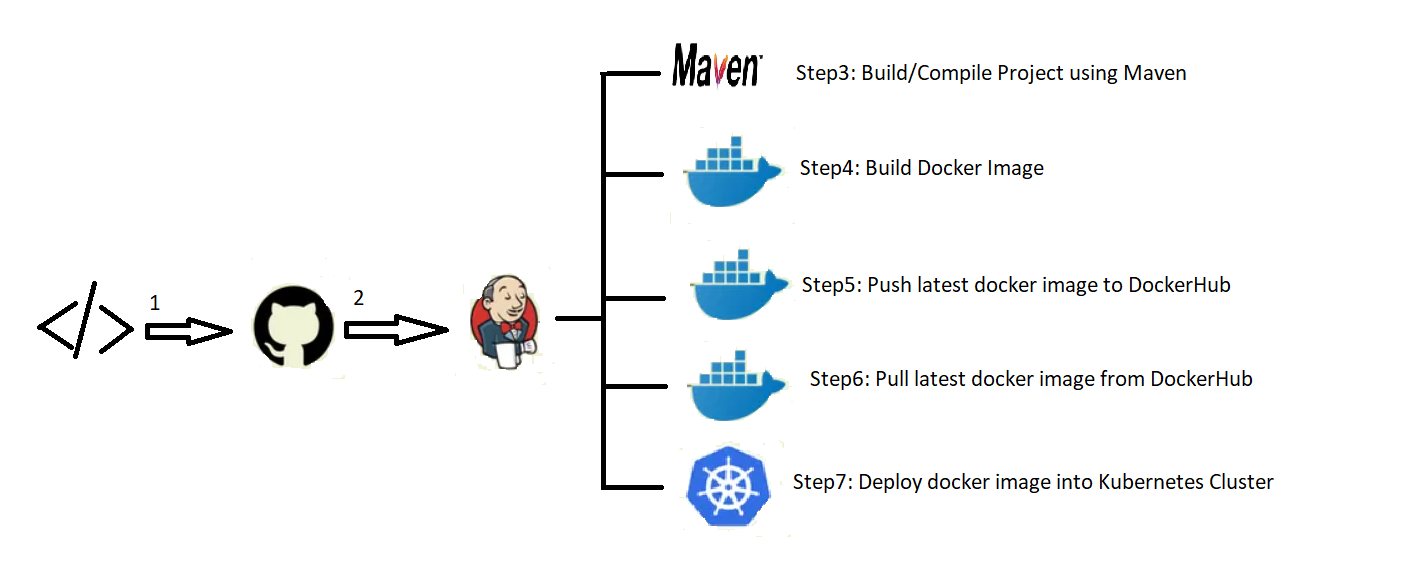
**Jenkins dashboard:** [**http://20.185.246.25:8080/job/springbootdemo**](http://20.185.246.25:8080/job/springbootdemo/)

**Login page of Shop me admin: http://20.185.246.25:32000/done**

Prerequisites:

1. Git
2. Jenkins
3. Docker
4. Kubernetes Cluster

Overview of the workflow:



Step 1: Checkin/Push your code to GitHub

Step2: Pull your code from github to jenkins server

Step3: Use Gradle/Maven build tool for building the artifacts

Step4: Create Docker Image

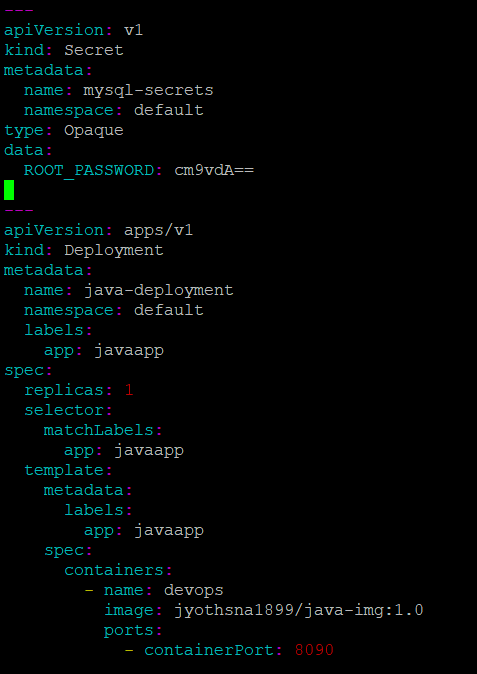
Step5: Push your latest Docker image to DockerHub

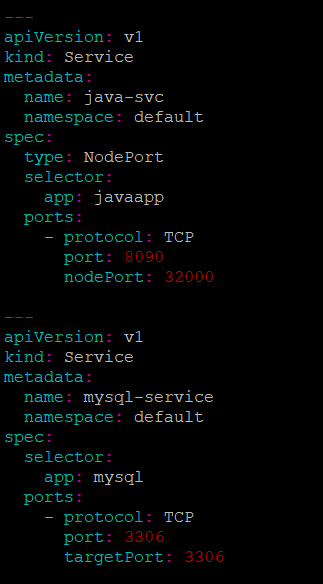
Step6: Pull the latest image from DockerHub into Jenkins

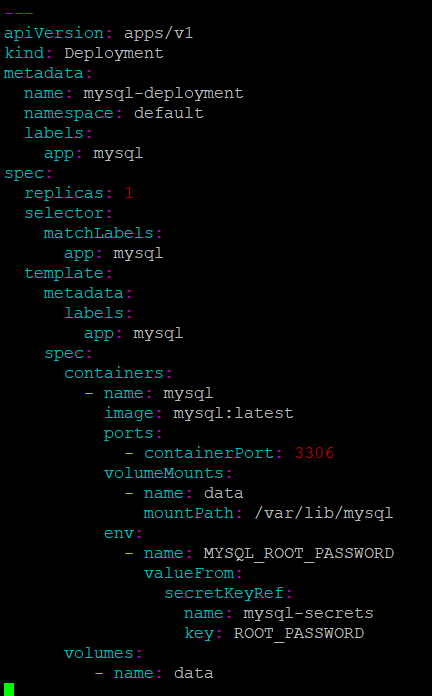
Step7: Then use springboot-app.yaml to deploy your application inside your Kubernetes Cluster

Create a yaml file for deploying the Application on Kubernetes using vi editor.

**$vi springboot-app.yaml**



-



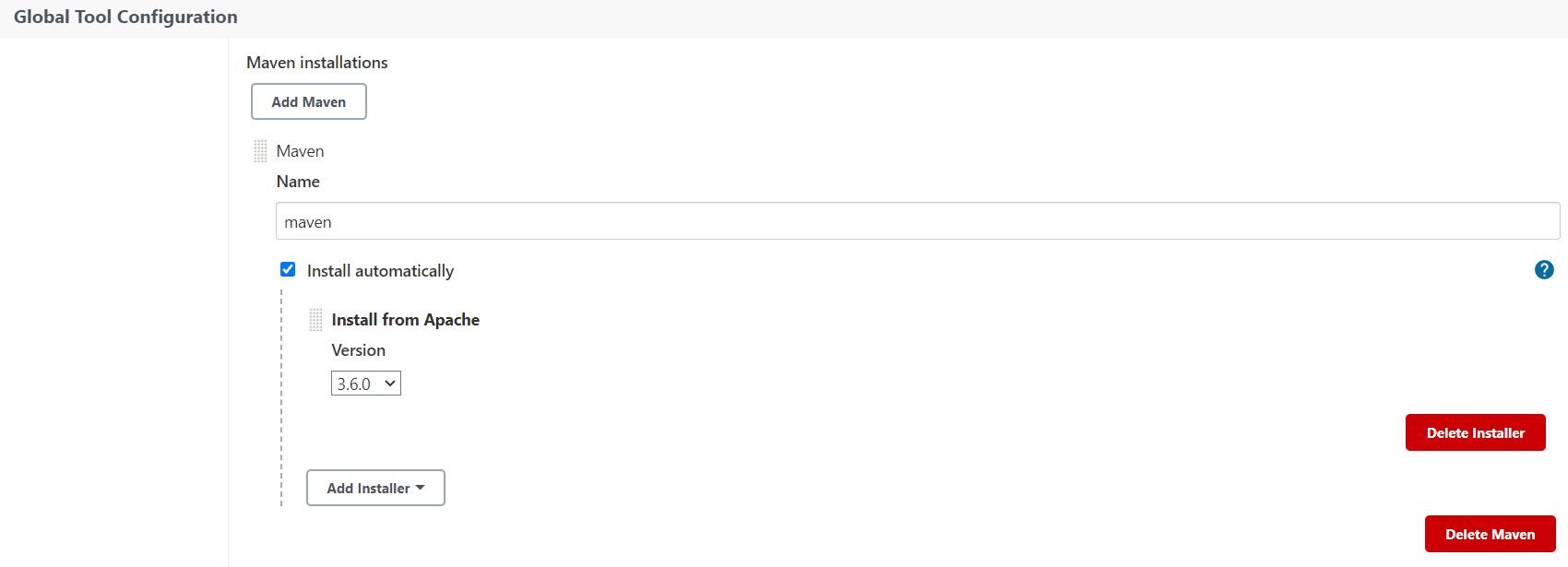
**Maven Setup:**

To setup Maven

Go to Manage Jenkins -> Global Tool Configuration -> Maven

Click on Add Maven and then enter the name Default.

After that click the checkbox Install Automatically and from the dropdown Iist select the latest version 3.6.0



Add current user to docker group:

The next step is to add the current user to Docker group to avoid the error-Got permission denied while trying to connect to the Docker Daemon socket at unix://var/run/docker.sock.

Use the below command for adding current user to Docker Group.

**$sudo usermod -aG docker $USER**

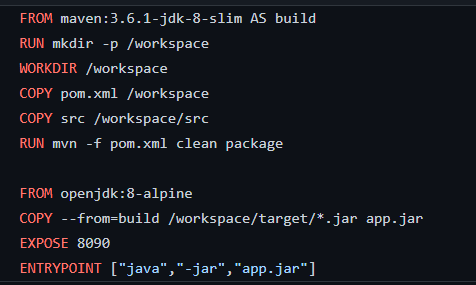
Add Jenkins User to Docker Group:

Similar to the previous step we also need to add Jenkins User to Docker Group, so that Jenkins can use Docker for building and pushing the docker images.

Use the below command to add Jenkins to Docker Group

**$sudo usermod -aG docker jenkins**

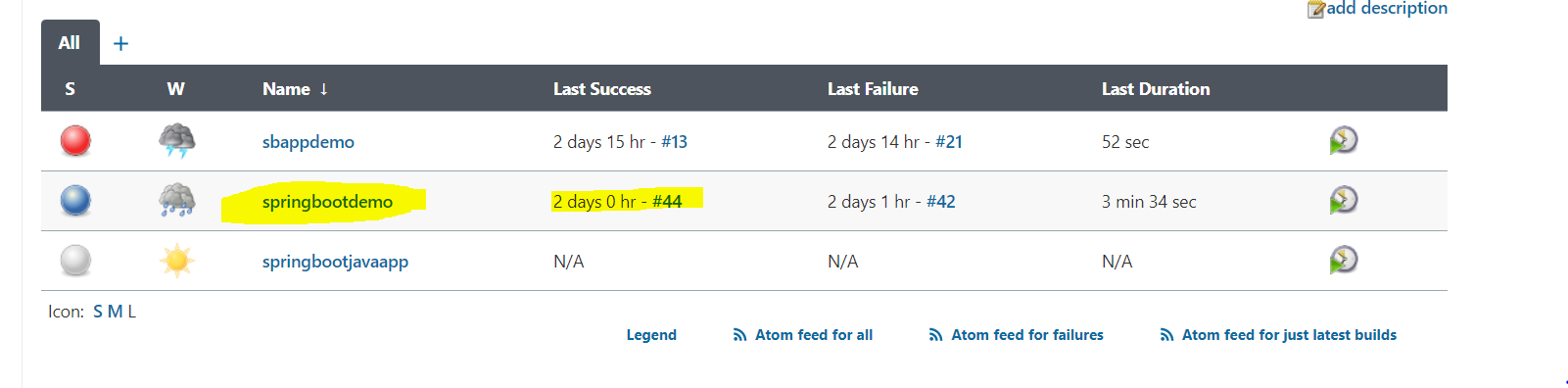
**Dockerfile for the spring boot application:**

****

**Steps for writing the pipeline script:**

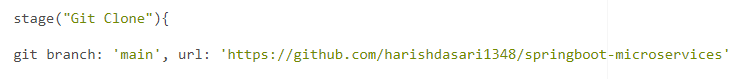
**Step1**: Create a Jenkins pipeline

1. Go to jenkins dashboard
2. Enter item name: springbootdemo
3. Select Pipeline
4. Click OK



**Step2:** Clone the Git Repo

The first principle of CI/CD pipeline is to clone/checkout the source code. So we are cloning git repo with jenkins

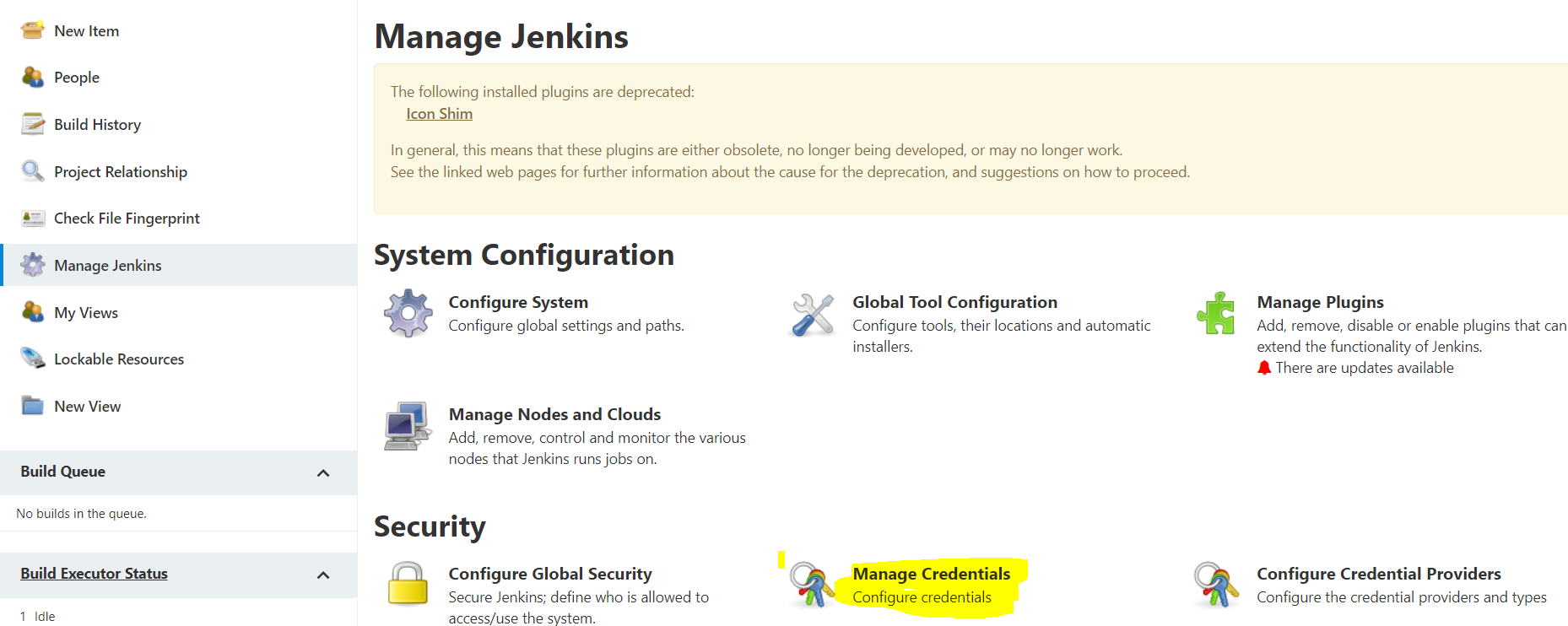


**Step3:** Jenkins store git credentials

We need to store the credentials in the case of a private git repository.

We cannot store plain text password inside jenkins scripts, so we need to store it somewhere securely. Jenkins manage credentials provides a way to store GitHub Username and Password.

Goto: Jenkins -> Manage Jenkins -> Manage Credentials





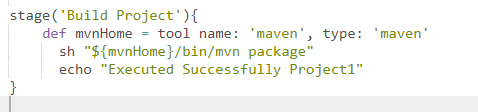
Then select username with password.

And then input your github username and password and ID.

Click ok.

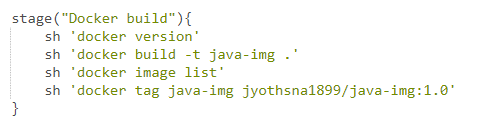
**Step4:** Build the spring boot application

Next step is to build the Spring Boot Application using Maven.



**Step5:** Build the docker image and tag it

After Maven build, build a docker image and then tag it with jyothsna1899/java-img:1.0



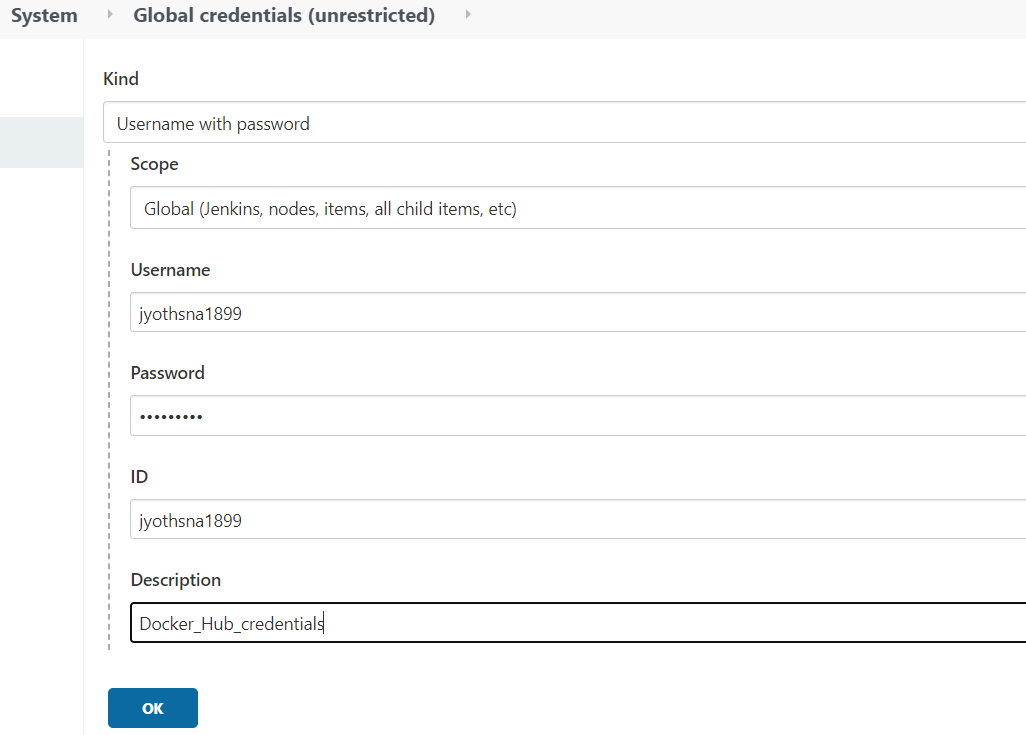
**Step6:** Jenkins store Docker Hub credentials

For storing DockerHub Credentials

Go to Jenkins -> Manage Jenkins -> Manage Credentials -> Stored scope to jenkins -> global -> Add credentials

From king dropdown select secret text.

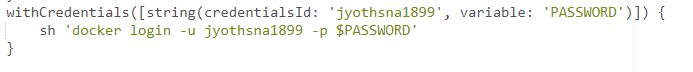
1. Secret- type in the DockerHub password
2. ID: DOCKER\_HUB\_PASSWORD
3. Description: dockerhub password



**Step7:** Docker login via CI

Since I am working inside Jenkins so every step I perform I need to write a pipeline script. Now after building and tagging the Docker Image we need to push it to the DockerHub. But before you push to DockerHub you need to authenticate yourself via CLI(command line interface) using docker login

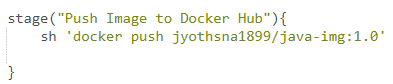
So here is the pipeline step for Docker Login



$DOCKER\_HUB\_PASSWORD - Since I cannot disclose my DockerHub password, so I stored my DockerHub Password into Jenkins Manage Jenkins and assigned the ID $DOCKER\_HUB\_PASSWORD

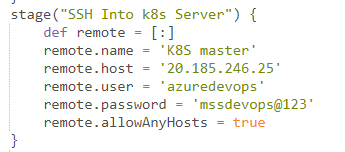
**Step8**: Push Docker image into DockerHub

After successful Docker login we need to push the image to DockerHub.



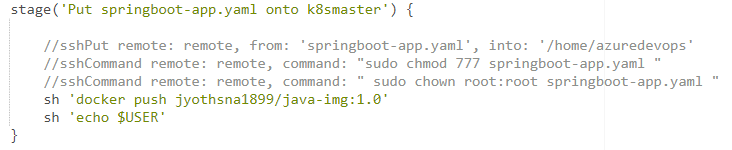
**Step9:** SSH Into k8smaster server

If you remember we have installed SSH Pipeline Steps in step no-5, now we are going to use that plugin to SSH into K8smaster server.



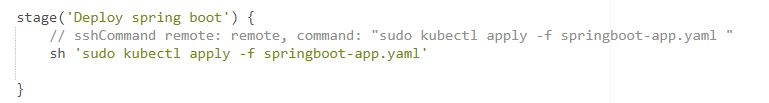
**Step10:** Copy k8s springboot-app.yaml to k8smaster server

After successful login copy k8s springboot-app.yaml into k8smaster server.



**Step11:** Create kubernetes deployment and service

Deploying springboot-app.yaml

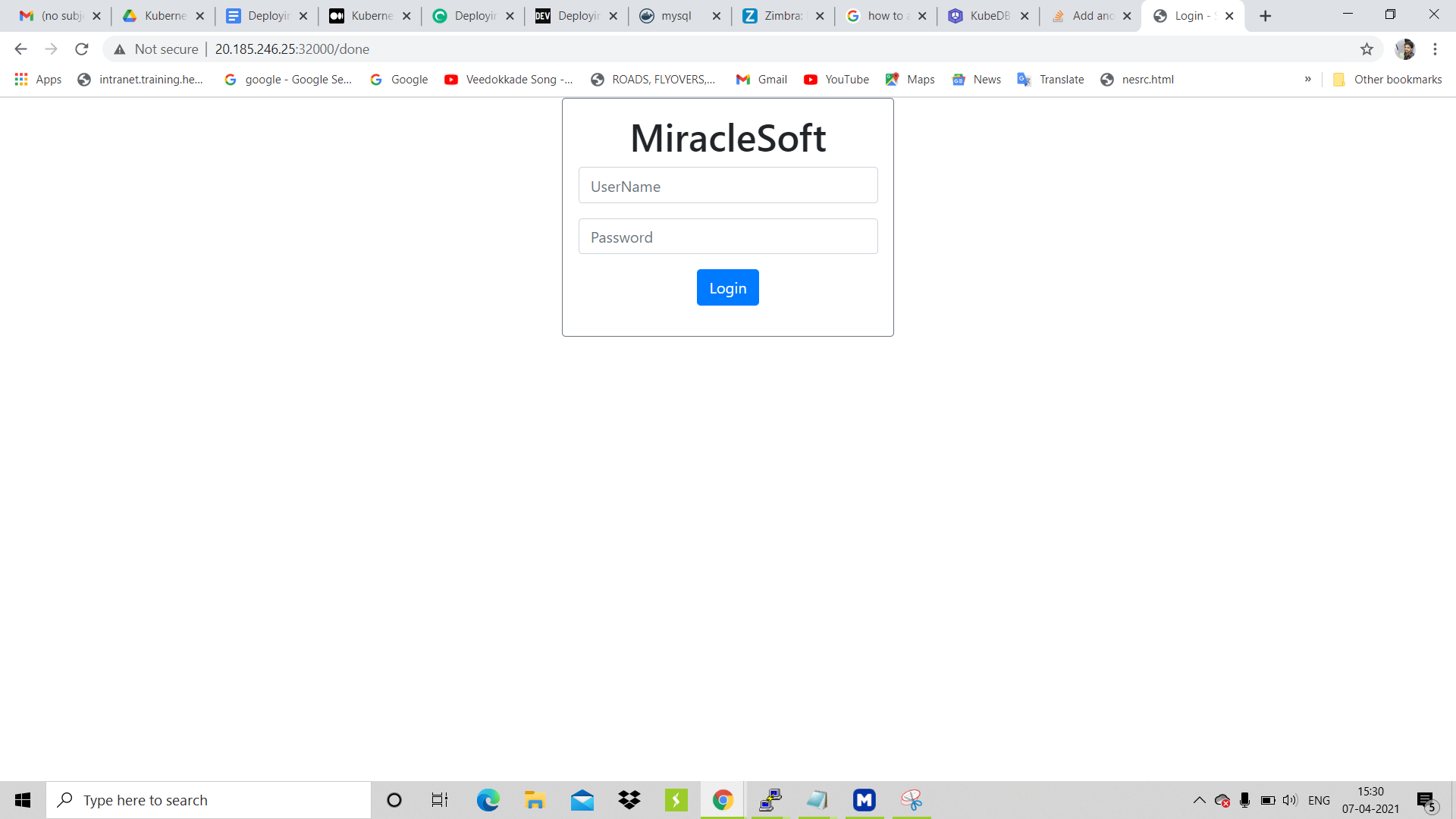


The below is the final pipeline script for CI/CD Pipeline

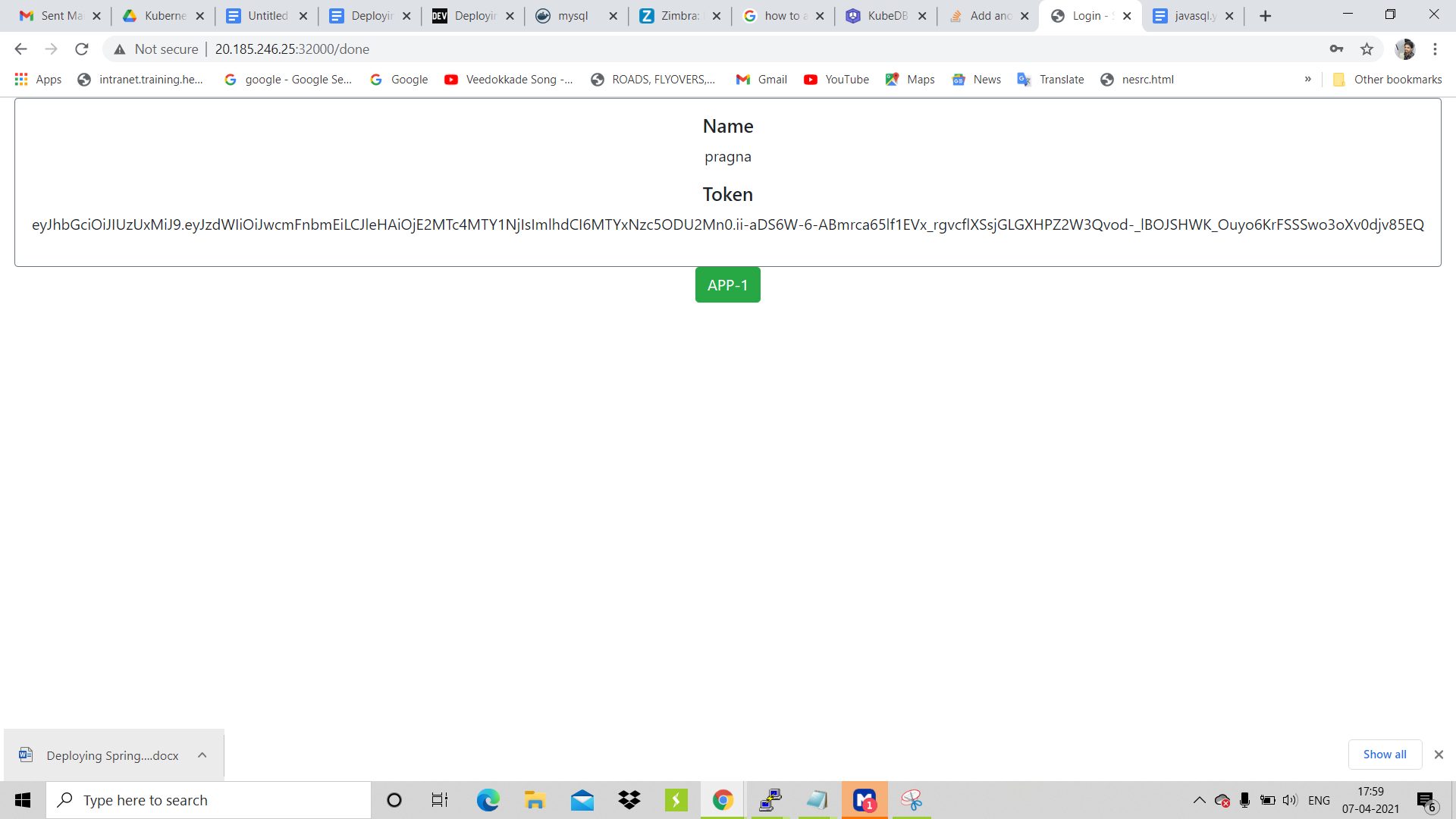


After building the jenkins pipeline the following login page will appear which can be accessed through the browser with below url.

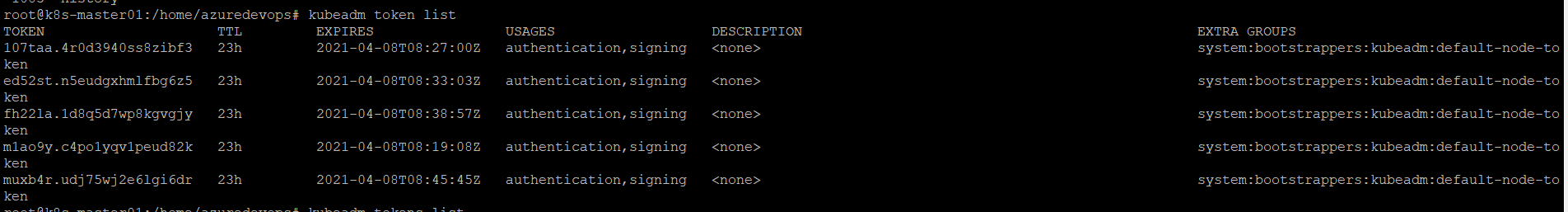
**http://20.185.246.25:32000/done**

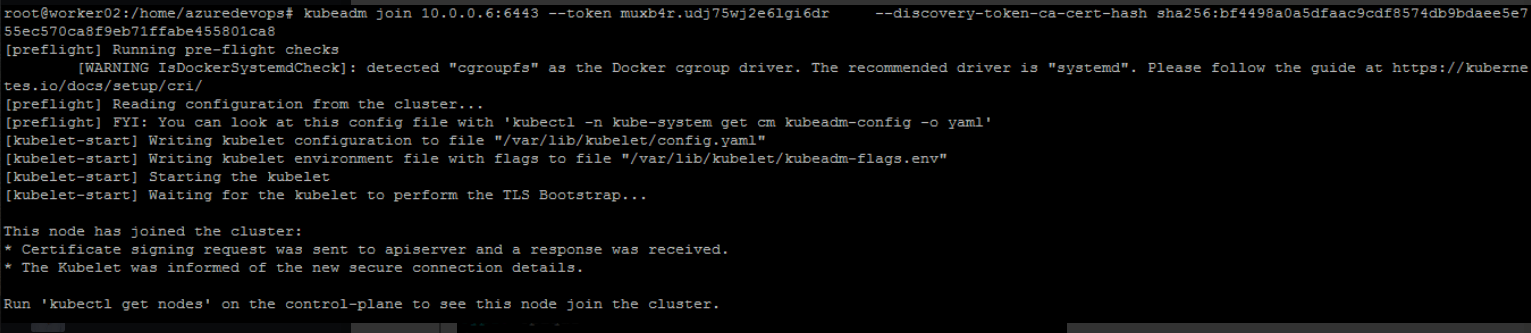


After the login page opens then login with user credentials and if login was successful then the below page will be displayed with the token.



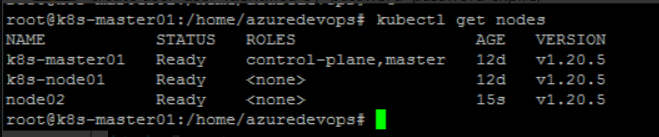
**Adding Worker node to the existing cluster:**

****

****

After adding the worker node to the cluster using the join command you can list all the nodes which are added to the cluster.

**$kubectl get nodes**

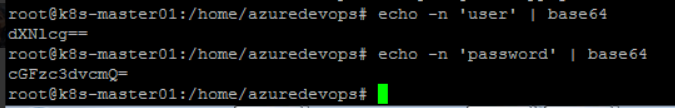


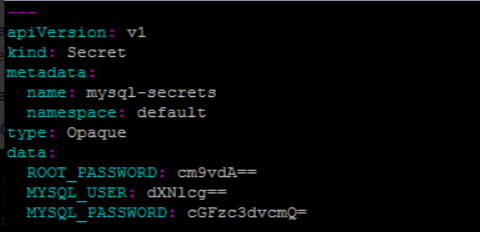
**Yaml files for Spring Boot Application:**

To deploy this application, we’ll use a few additional concepts in Kubernetes called PersistentVolumes, PersistentVolumeClaim, Secrets,Deployments,Services,Configmaps./

**Secrets:**

We’ll make use of Kubernetes secrets to store the Database credentials. A Secret is an object in Kubernetes that lets you store and manage sensitive information, such as passwords, tokens, ssh keys etc. The secrets are stored in Kubernetes backing store, etcd. You can enable encryption to store secrets in encrypted form in etcd.





**Persistent Volumes and Persistent Volume Claim:**

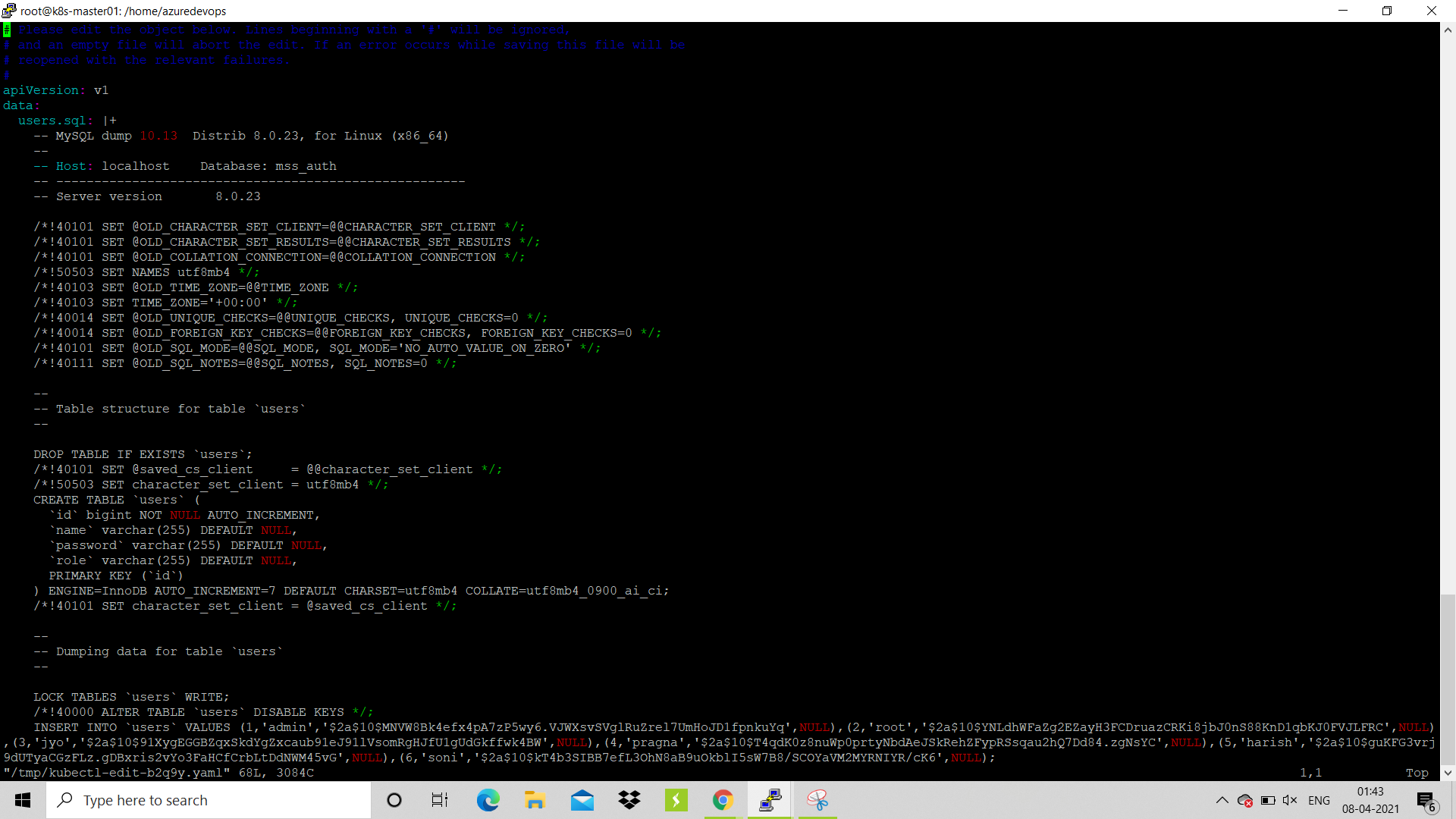
We’ll use Kubernetes Persistent Volumes to deploy Mysql. A PersistentVolume (PV) is a piece of storage in the cluster. It is a resource in the cluster just like a node. The Persistent volume’s lifecycle is independent from Pod lifecycles. It preserves data through restarting, rescheduling, and even deleting Pods.

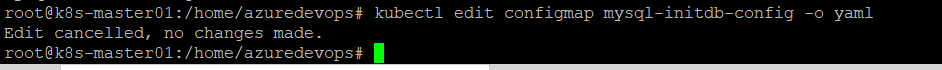
PersistentVolumes are consumed by something called a PersistentVolumeClaim (PVC). A PVC is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). PVCs can request specific size and access modes (e.g. read-write or read-only).



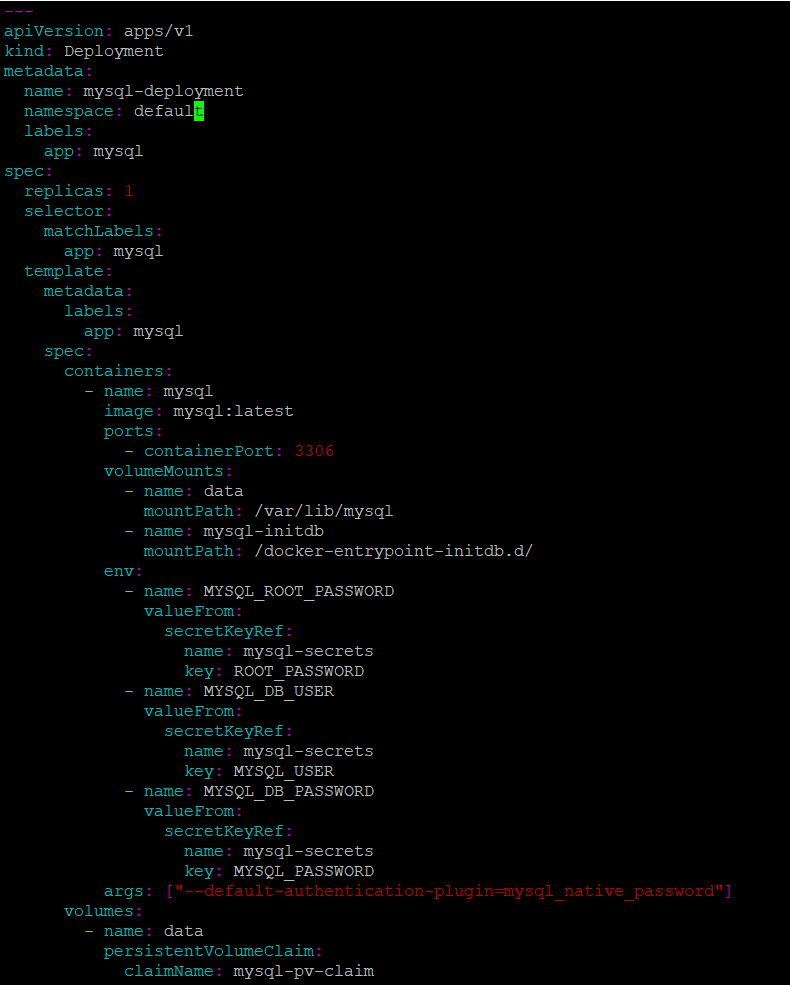
We used ConfigMap:

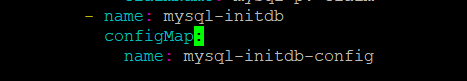
**$kubectl edit configmap mysql-initdb-config -o yaml**

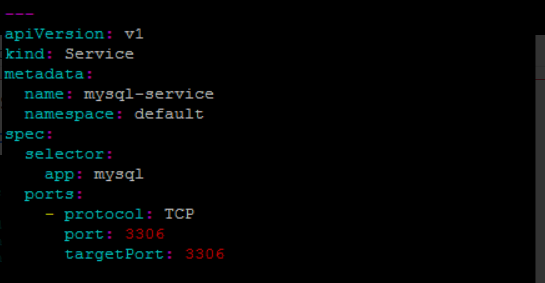




**Deploying Mysql on Kubernetes using PersistentVolume and Secrets**



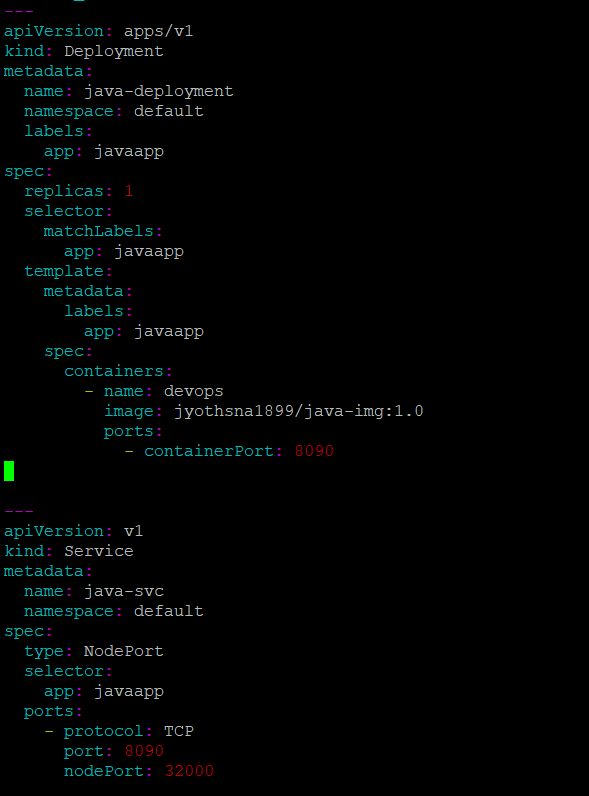




We’re creating four resources in the above manifest file. A PersistentVolume, a PersistentVolumeClaim for requesting access to the PersistentVolume resource, a service for having a static endpoint for the MySQL database, and a deployment for running and managing the MySQL pod.

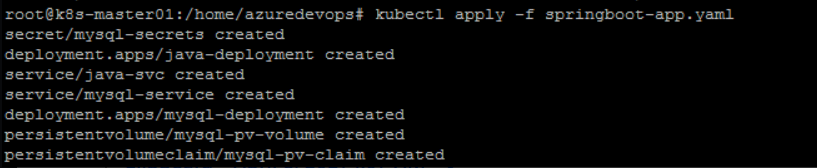
The MySQL container reads database credentials from environment variables. The environment variables access these credentials from Kubernetes secrets.

## **Deploying Spring Boot Application on Kubernetes:**

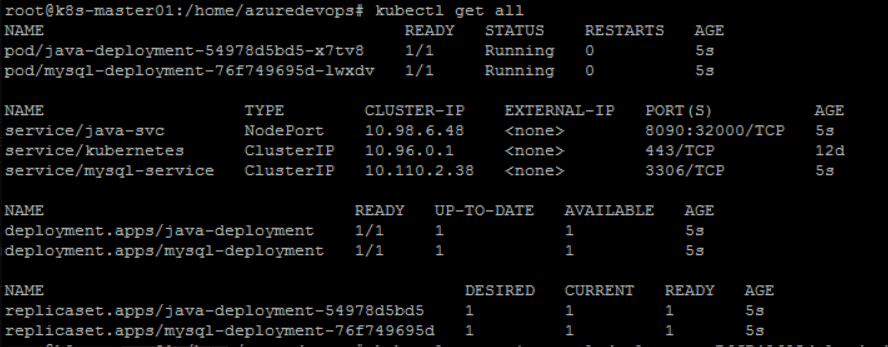


To create a deployment,service, PV, PVC, Secret we used the below command.

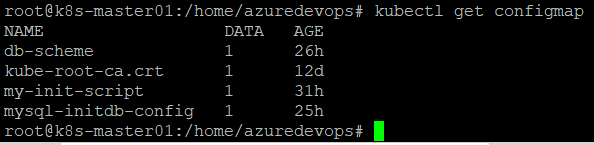
**$kubectl apply -f springboot-app.yaml**



**$kubectl get all**



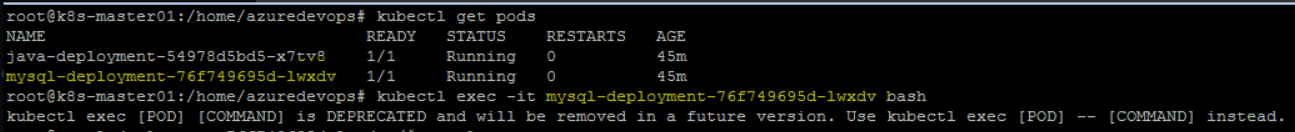
**$kubectl create configmap mysql-initdb-config --from-file=users.sql**



**Logging into the MySQL pod:**

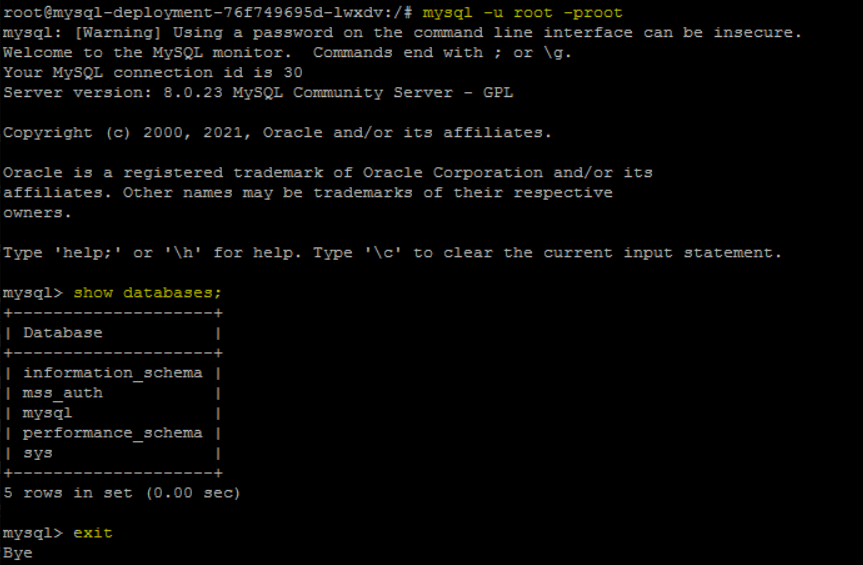
You can get the MySQL pod and use kubectl exec command to login to the Pod

**$kubectl exec -it mysql-deployment-76f749695d-1wxdv bash**



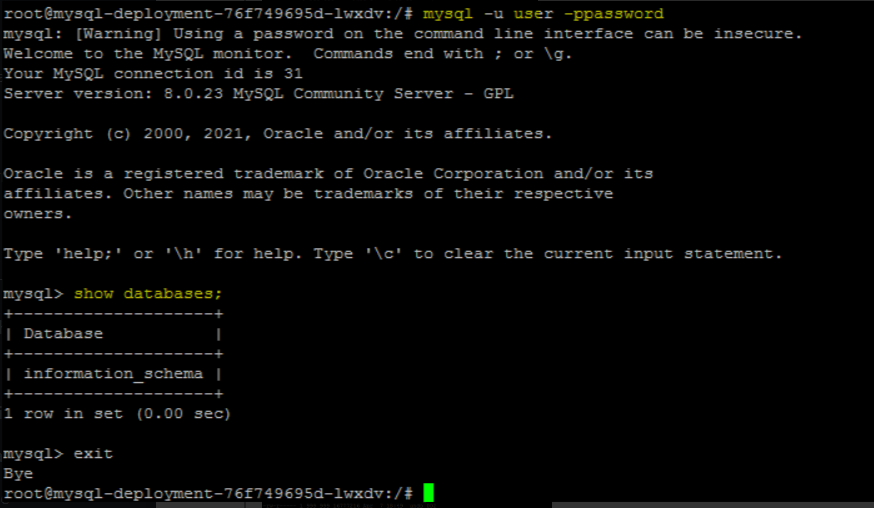
After logging into the pod you will get a hash prompt. There you can give the below command to login to mysql as a root user.

**#mysql -u root -proot**

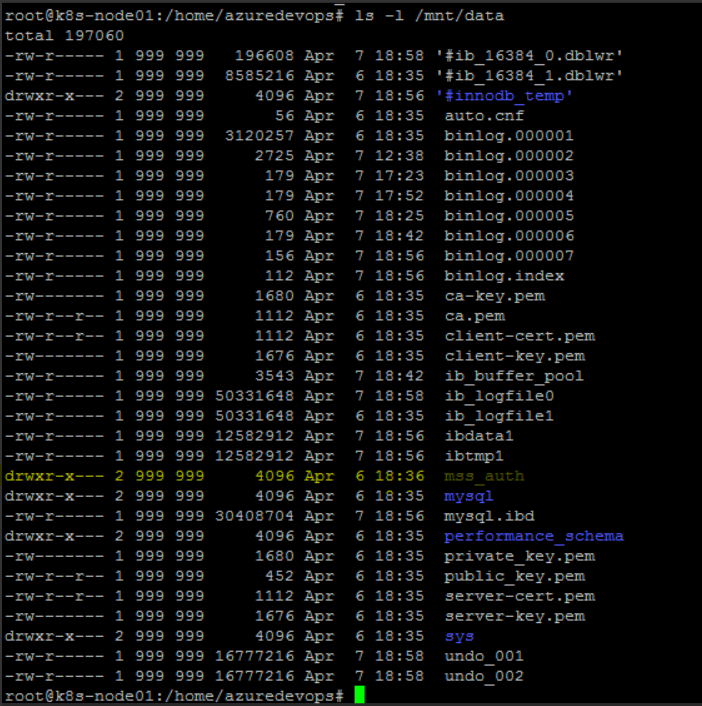


**Logging into mysql as a new user:**

**#mysql -u user -ppassword**



A directory will be created with the name of **mss\_auth** when a deployment is created.



Also a container is created and it is up and running as shown below.

Containers in node01



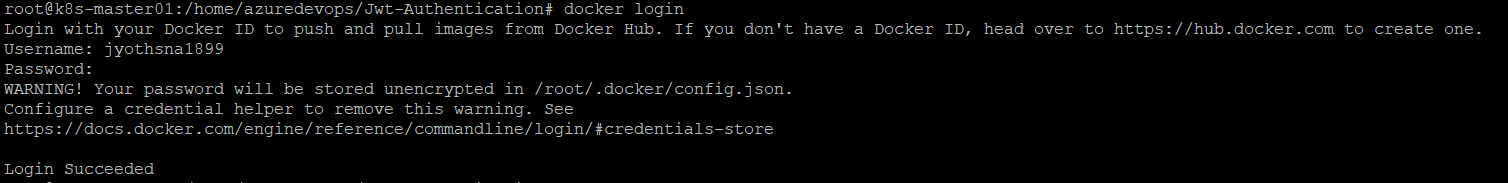
Containers in node02



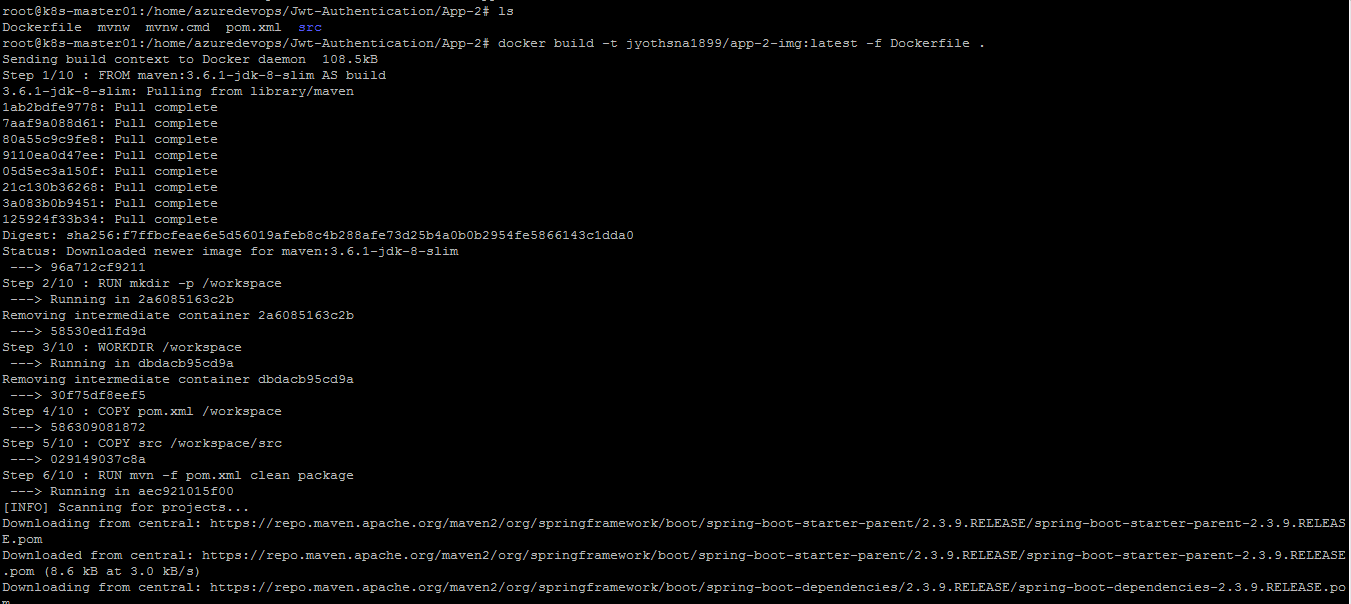
**Deploying SpringBoot Application with Nodejs on Kubernetes**

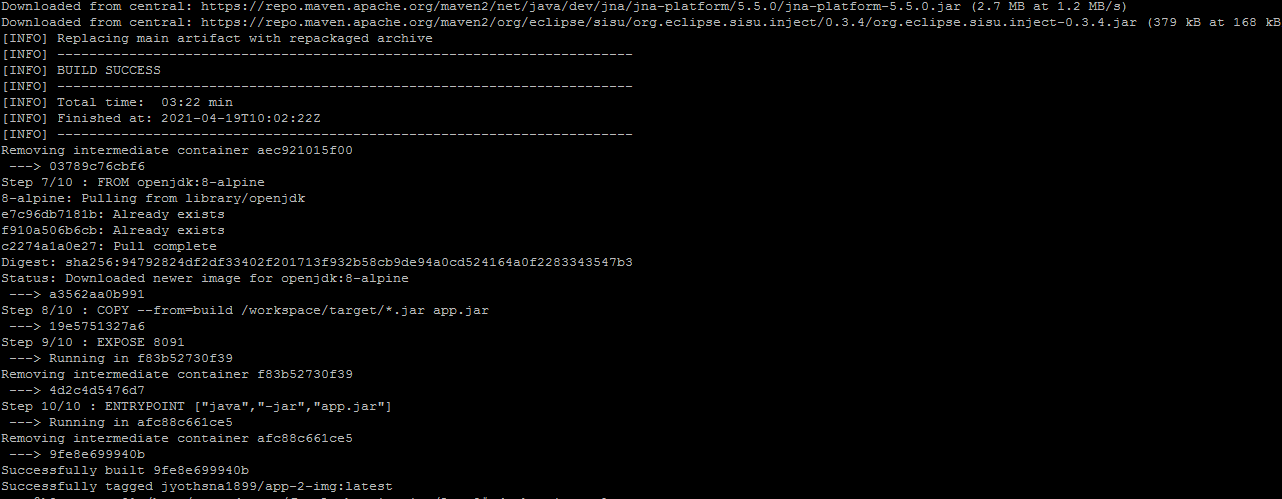
Github Url:

**$docker login**

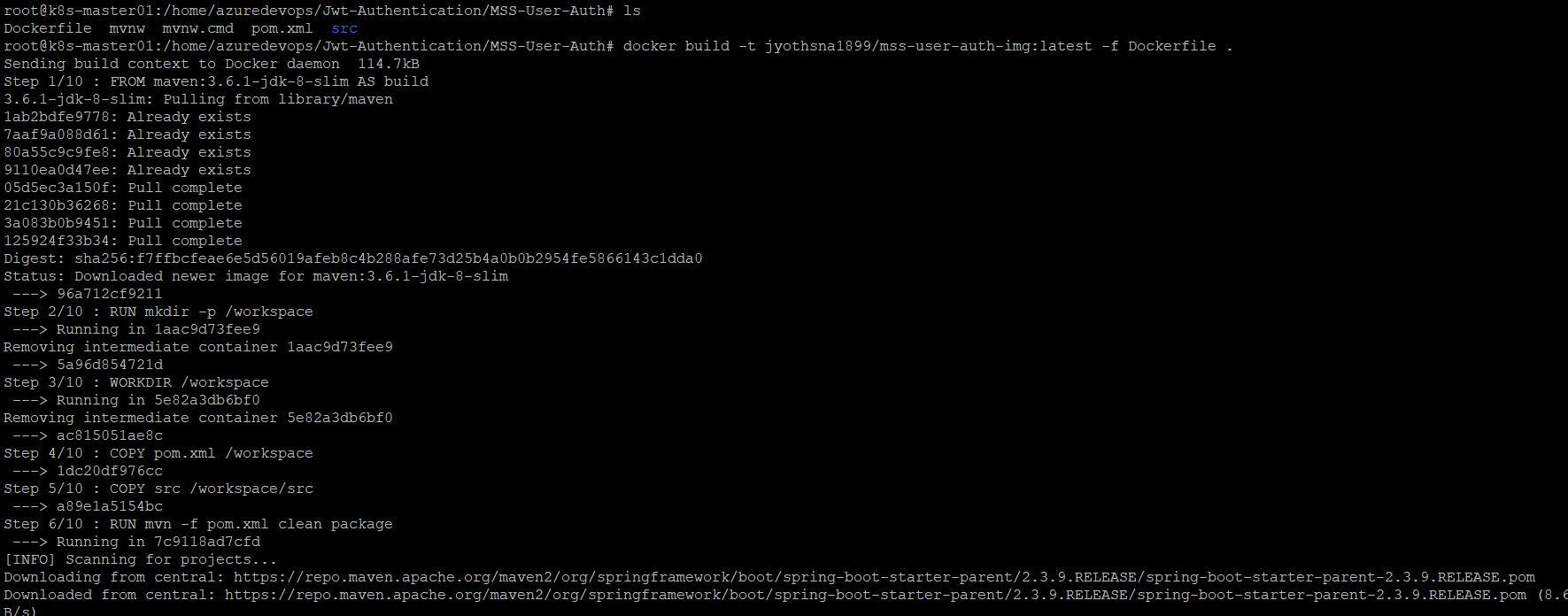


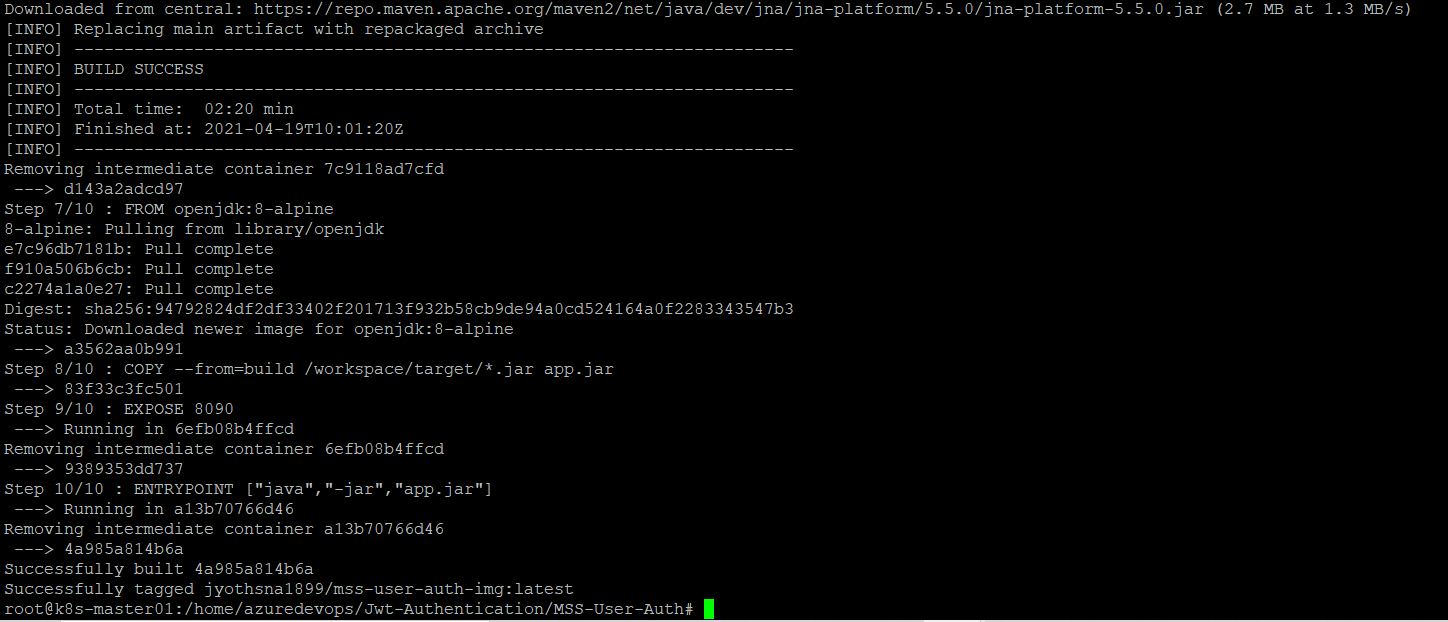
$**docker build -t jyothsna1899/app-2-img:latest -f Dockerfile .**



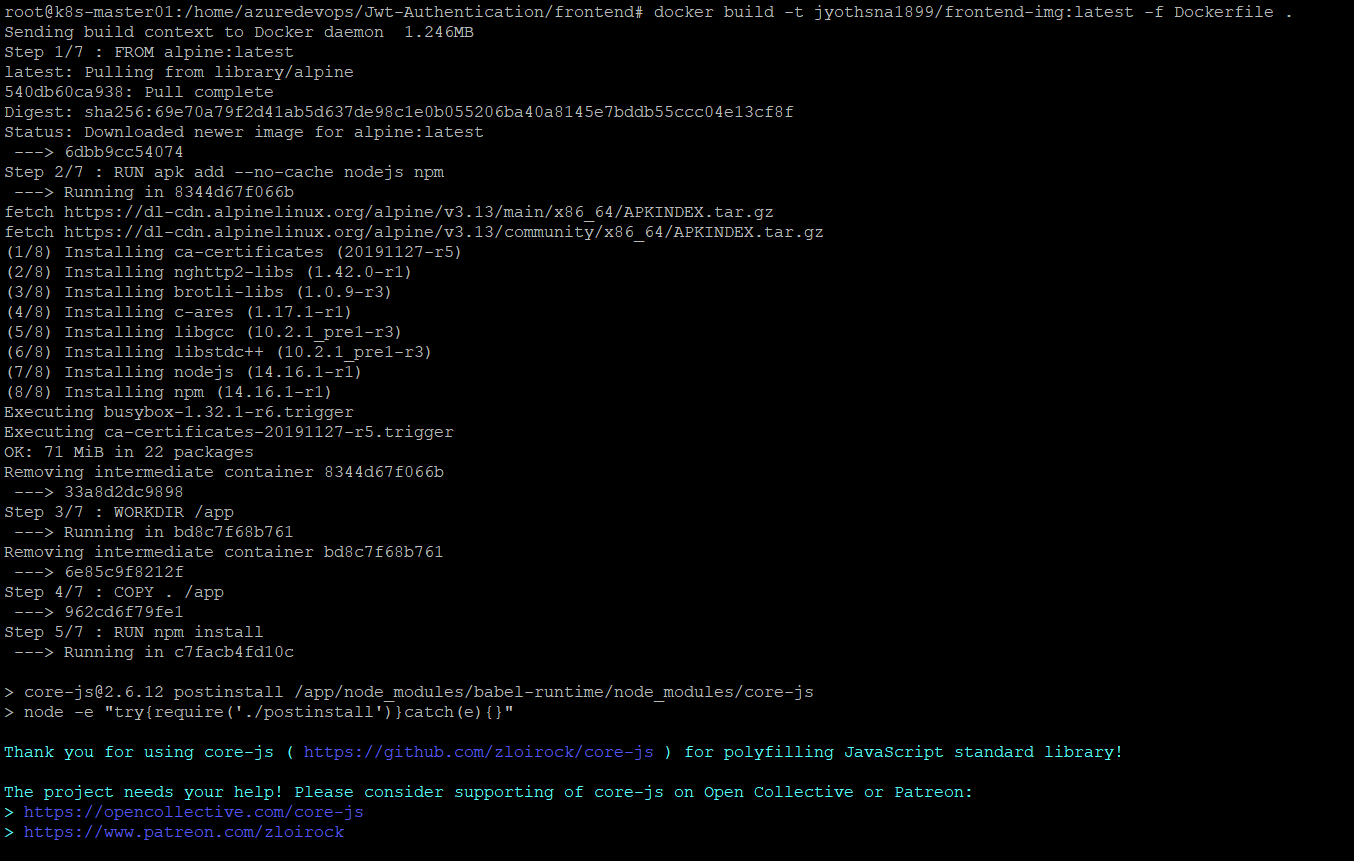


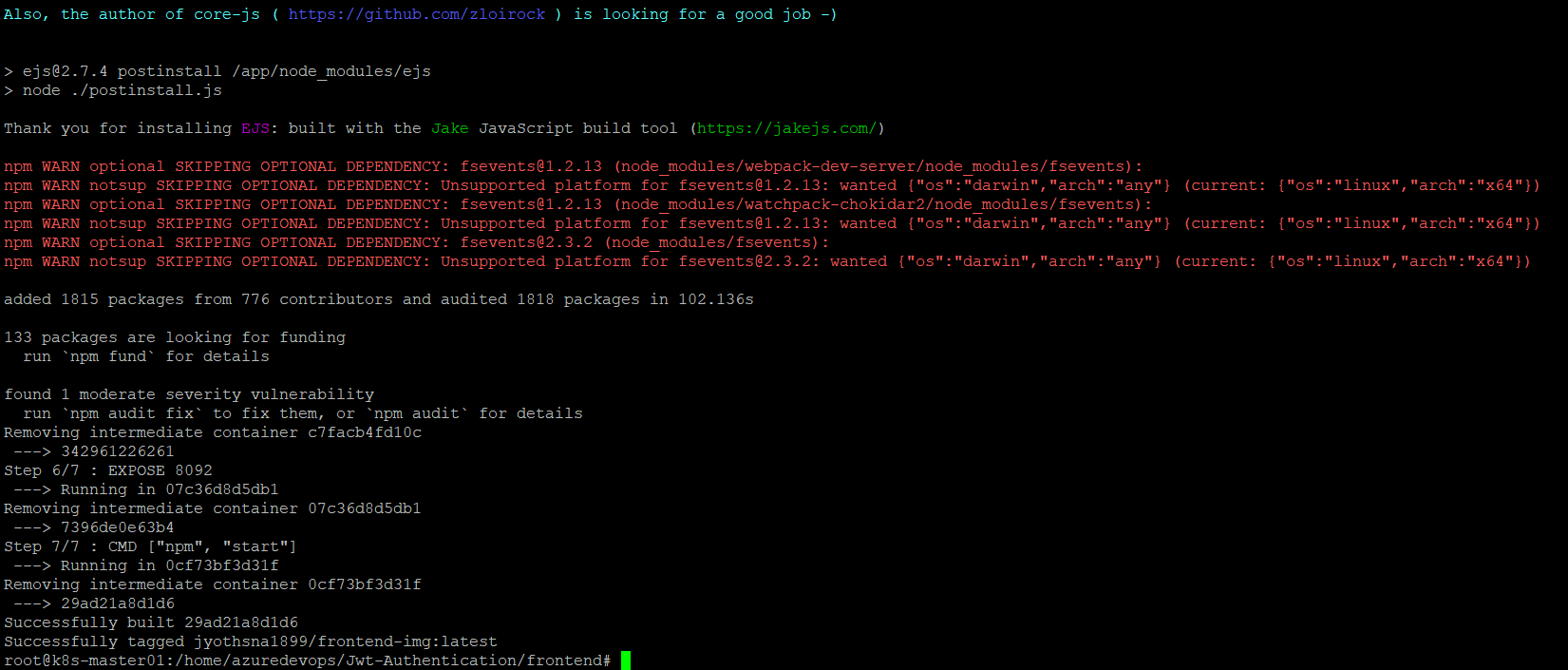
**$docker build -t jyothsna1899/mss-user-auth-img:latest -f Dockerfile .**



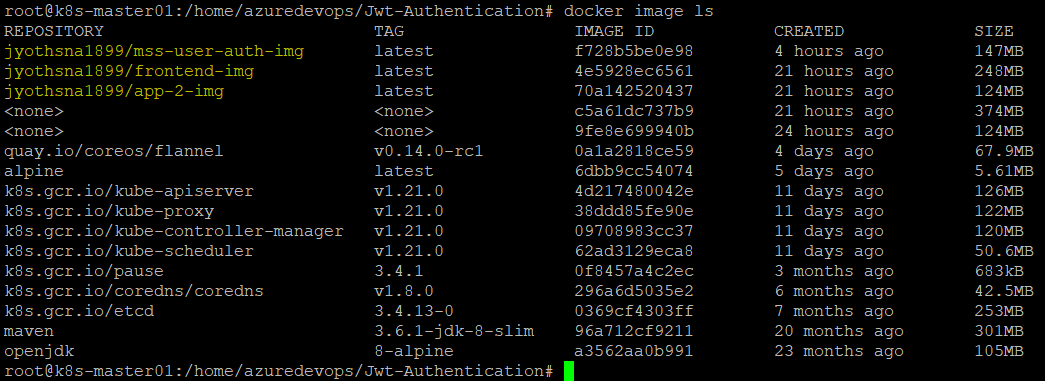


**$docker build -t jyothsna1899/frontend-img:latest -f Dockerfile .**

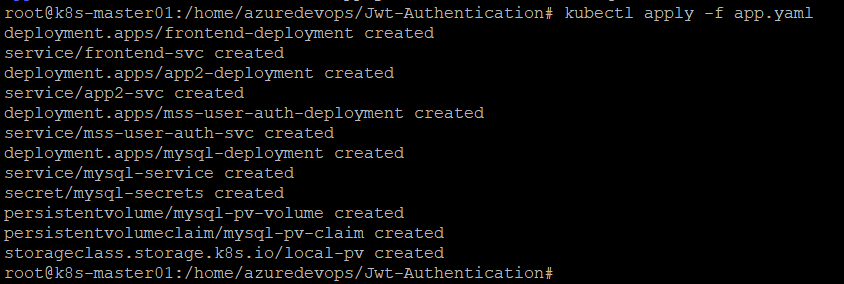




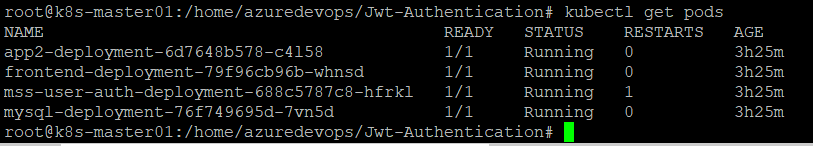
**$docker image ls**

****

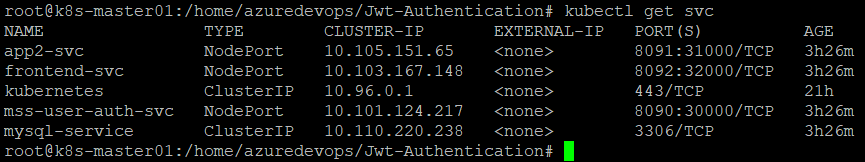
**$kubectl apply -f app.yaml**



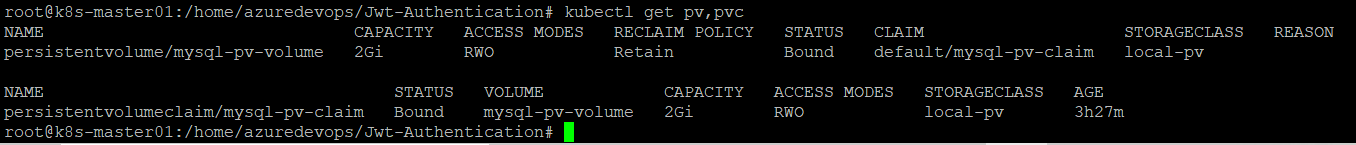
**$kubectl get pods**



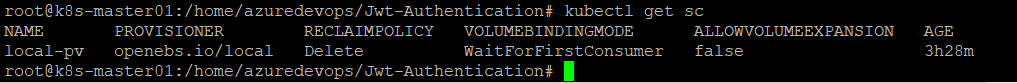
**$kubectl get svc**



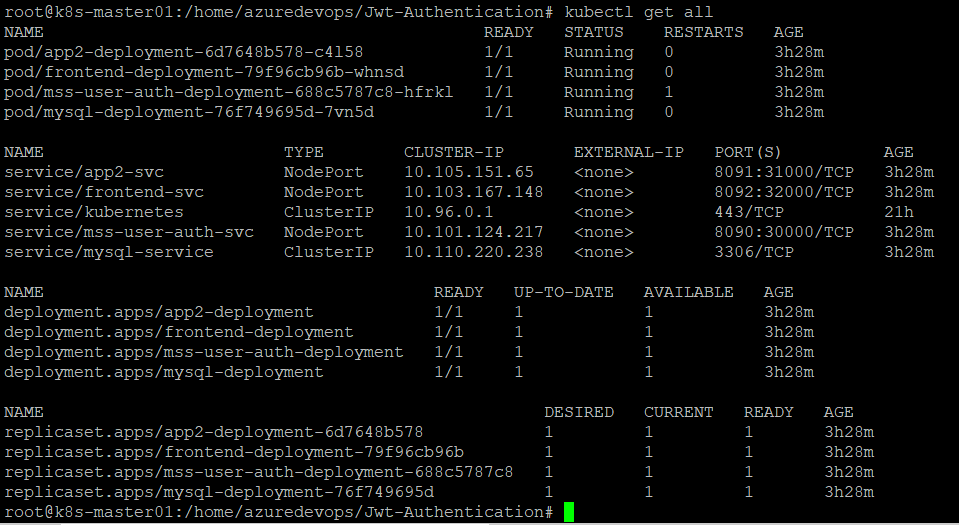
**$kubectl get pv,pvc**



**$kubectl get sc**

****

**$kubectl get all**



Containers running in worker node

**$docker ps**



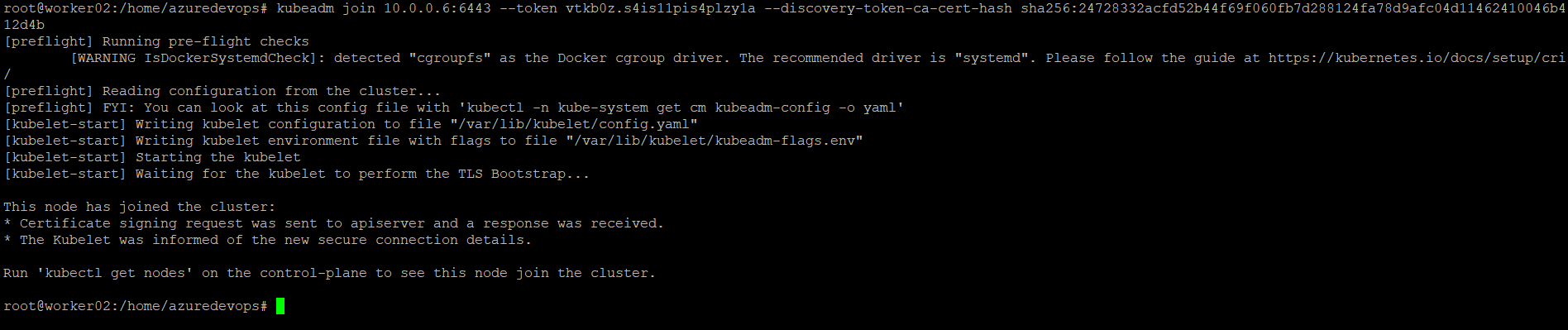
**Adding new Worker node to the cluster**

**In Master node**

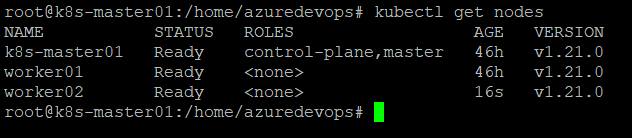
**$kubeadm token create --print-join-command**

****

**In Worker node**

****

**$kubectl get nodes**

****